Eduardo Rafael da Silva Vieira Frederico Marques

Licenciado em Engenharia Informática

Single Operation Multiple Data - Paralelismo de Dados ao Nível da Sub-rotina

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Hervé Miguel Cordeiro Paulino,
Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Doutor João Carlos Gomes Moura Pires
Arguente:	Prof. Doutor Marcelo Pasin
Vogal:	Prof. Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2012

Single Operation Multiple Data - Paralelismo de Dados ao Nível da Sub-rotina

À minha família

Agradecimentos

Em primeiro lugar deixar o meu agradecimento ao meu orientador, Professor Hervé Paulino, pelo apoio dado durante a realização desta dissertação, assim como na elaboração do artigo, e também ao Professor Artur Miguel Dias, pela ajuda que forneceu na introdução à ferramenta Polyglot.

Agradeço também a todos os meus colegas que se cruzaram comigo neste percurso académico. Em especial queria agradecer ao João Saramago e Valter Balegas pelos momentos de boa disposição que proporcionaram ao longo deste semestre da elaboração.

E mais importante que tudo, agradeço aos meus pais, irmão e restantes familiares pelo apoio total que me deram ao longo da vida.

Resumo

O advento dos processadores *multi-core* marcou o nascimento de uma nova era, em que se requer que as aplicações sejam concorrentes para que possam usufruir da natureza paralela do *hardware*. No entanto, esta mudança de paradigma na arquitectura dos processadores não foi acompanhada por alterações significativas nas linguagens de programação de uso generalizado. Obrigando frequentemente o programador a optar entre desempenho (com o recurso a computação paralela) e a produtividade no processo de desenvolvimento do software.

Neste contexto, esta dissertação propõe a aplicação do paradigma de paralelismo de dados ao nível da sub-rotina. A invocação de uma sub-rotina dá origem a várias tarefas, cuja execução opera sobre partições distintas dos dados de entrada. A execução destas tarefas é delegada a uma *pool* de *threads* trabalhadores, que as executarão em paralelo, segundo uma variação do modelo de execução *Single Program Multiple Data* a que baptizámos como *Single Operation Multiple Data*. Este modelo é apresentado ao programador segundo o paradigma *Distribute-Map-Reduce*, em que os dados de entradas são particionados e submetidos às múltiplas instâncias da sub-rotina para execução paralela. Aos resultados parciais é posteriormente aplicada uma operação de redução para calcular o resultado final.

A instanciação do modelo foi realizada como uma extensão à linguagem de programação Java, sendo o sistema de execução construído sobre o sistema de execução da linguagem X10 [CGS+05]. O protótipo resultante pode ser aplicado tanto em ambientes de memória partilhada como distribuída.

A avaliação realizada atesta a viabilidade da solução, apresentando resultados de desempenho interessantes para um conjunto considerável de aplicações, sem que o programador tenha de escrever código especializado.

Palavras-chave: Computação paralela, Paralelismo de dados, Modelo de execução SOMD, *Distribute-Map-Reduce*

Abstract

Multi-core processors inaugurated a new era in CPU design and organization, that requires applications to be concurrent, in order to fully benefit from the parallel nature of the hardware. However, this paradigm shift in processor architecture was not followed by significant changes in mainstream programming languages. Therefore, currently software developers must often choose between performance (by resorting to parallel computing) and productivity in the software development process. This state of the art makes this area of great relevance and impact in current computer science research.

In this context, this dissertation proposes the application of the data parallel paradigm at subrotine level. The invocation of a subrotine spawns several tasks, each operating upon a partition of the input dataset. The execution of these tasks is delegated to a pool of worker threads that will execute them in parallel, accordingly to a variant of the execution model *Single Program Multiple Data* which we baptised *Single Operation Multiple Data*. This model is presented to the programmer as a *Distribute-Map-Reduce* paradigm, where the data of the input dataset is decomposed and submitted to multiple instances of the subrotine that are executed in parallel. The partial results are after subtmitted to a reduction operation, which will compute the final result.

The model was instantiated as an extension to the Java programming language, supported by a runtime system built on top of the Java runtime for the X10 programming language [CGS+05]. The resulting prototype is able to execute applications in both shared and distributed memory environments.

The performed evaluation attest the viability of the solution. We obtained good performance results for a considerable set of applications without burdening the programmer with the writing of specialized code.

Keywords: Parallel Programming, Data Parallelism, SOMD execution model, Distribute-Map-Reduce.

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

1

Introdução

1.1 Motivação

Em 2006 verificou-se uma importante mudança no paradigma de construção dos processadores para computadores pessoais, a transição de processadores *single-core* para processadores *multi-core*. A Lei de Moore, que constata que o número de transístores num processador duplica a cada dois anos sem custos acrescidos, era acompanhada, até então, por um aumento na velocidade do relógio, o que permitia que um programa sequencial fosse cada vez mais rápido com a evolução do processador. Contudo, apesar da lei ainda se manter válida, limites físicos que têm como consequência o sobreaquecimento do processador, impediram que o aumento da velocidade de relógio pudesse manter-se como o motor da evolução dos processadores. Neste contexto, a indústria de processadores foi obrigada a optar por outras alternativas, nomeadamente a introdução de múltiplos processadores num só *chip* - as arquitecturas *multi-core* - e o aumento significativo da capacidade da memória *cache*.

Esta mudança de paradigma arquitectural teve um impacto significativo no desempenho das aplicações sequenciais. O aumento do número de *cores* não se traduz em melhorias de desempenho nesta categoria de aplicações. Podendo até haver um decréscimo de desempenho devido à competitividade pelos recursos por parte dos *cores*. Com este novo paradigma apenas aplicações concorrentes podem tirar benefício destas arquitecturas, vendo o seu desempenho aumentar com o número de processadores.

No entanto, apesar dos processadores *multi-core* já existirem desde 2006, as linguagens de programação de uso generalizado, compiladores e sistemas de execução, permanecem na sua generalidade inalteradas, não tendo sofrido alterações significativas com vista a adaptarem-se a este novo tipo de arquitecturas de processadores. A comunidade almeja

assim que sejam propostas novas soluções ao nível de linguagens, compiladores e sistemas de execução para que se possa tirar melhor partido do paralelismo presente neste tipo de processadores.

O desafio desta área é encontrar construções linguísticas, quer a nível de construtores de linguagem ou de bibliotecas. Estes têm como função permitir abstrair o programador dos vários detalhes inerentes à programação paralela, nomeadamente, composição funcional e de dados, e o seu mapeamento na arquitectura alvo. Estas construções deverão ter uma curva de aprendizagem pequena, para que possam obter uma produtividade nas linguagens de computação paralela semelhante à produtividade observada na programação de uso geral. Este aumento na produtividade permitirá então trazer a programação paralela para o desenvolvimento de aplicações do dia-a-dia, podendo ser usada por não-especialistas na área da computação paralela.

Neste contexto esta dissertação propõe um modelo de execução, que permite expressar paralelismo de dados de forma simples ao nível de uma sub-rotina.

1.2 Single Operation Multiple Data

A decomposição paralela de um problema pode incidir sobre os seus dados (paralelismo de dados) ou sobre o seu código (paralelismo de tarefas ou funcional). Este último é normalmente explorado através de sub-rotinas que são submetidas para execução a um conjunto *threads* organizados em *pools*. Este conceito é a base do modelo de execução de linguagens como o Cilk [FLR98], ou o X10 [CGS+05] ou os executores do Java (incluídas no pacote java.util.concurrent[1]). Já o paralelismo de dados é normalmente explorado ao nível do ciclo, sendo exemplos desta abordagem o OpenMP [DM98], o Intel TBB [Rei07] e linguagens PGAS (*Partitioned Global Address Space*) como o UPC [UPC05] ou o X10.

Esta dissertação tal como referido anteriormente, propõe expressar o paralelismo de dados ao nível de sub-rotinas. A invocação duma sub-rotina neste contexto inicia a criação de várias tarefas, que trabalharão sobre diferentes partições dos dados de entrada, aplicando o mesmo conjunto de operações a cada uma dessas partições. Estas tarefas serão executadas por múltiplos *threads* trabalhadores, que executarão em conformidade com uma variação do modelo de execução *Single Program Multiple Data* (SPMD) [DGNP88], que baptizámos como *Single Operation Multiple Data* (SOMD).

O modelo de execução é apresentado ao programador segundo o paradigma de programação *Distribute-Map-Reduce* (DMR), no qual o conteúdo dos dados de entrada é particionado e alimentado às múltiplas instâncias da sub-rotina (MI) que executam em paralelo. Aos resultados parciais obtidos da execução das MIs é aplicada uma função de redução, que calcula o resultado a ser devolvido ao invocador. A figura 1.1 ilustra o modelo, onde se pode observar que várias tarefas executam em paralelo sobre partições de dados diferentes, produzindo, cada uma delas, um resultado parcial. O cálculo do resultado final é realizado aplicando uma função de redução aos resultados parciais.

[1] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

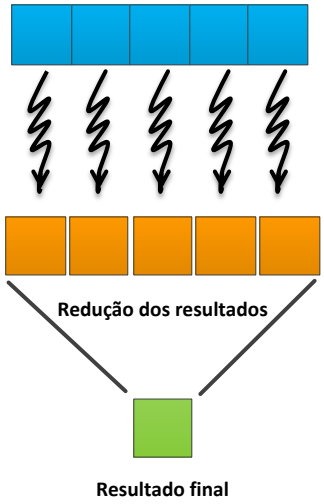


Figura 1.1: Execução do modelo SOMD

Esta abordagem permite ao programador comum expressar computações paralelas através da anotação de sub-rotinas inalteradas. A simples indicação das estratégias de distribuição e redução é suficiente para que, partes da aplicação (a sub-rotina anotada) execute em paralelo, permitindo assim tirar partido da natureza paralela do ambiente de execução alvo, sem a criação de código especializado para o efeito.

Como será observado nos exemplos apresentados ao longo deste documento, as distribuições e reduções são normalmente independentes de uma sub-rotina em particular e portanto podem ser utilizadas em diversos casos. Nesta medida antevemos que estas possam ser disponibilizadas sob a forma de bibliotecas. Nesse caso, a tarefa do programador restringe-se à selecção das políticas de distribuição e redução a aplicar. Porém o programador mais conhecedor da área poderá sempre criar as suas próprias políticas, de modo a particionar outras estruturas de dados, ou criar partições e funções de redução especializadas. O programador pode ainda modificar o código da sub-rotina alvo de modo a mapear melhor no modelo de execução SOMD, seja por questões de desempenho ou algorítmicas.

Esta dissertação foca a incorporação do modelo SOMD na linguagem de programação Java, através da criação duma extensão que introduz construtores que expõem o paradigma de programação DMR. A extensão de linguagem foi criada usando a ferramenta *Polyglot* [NCM03], através da geração dum compilador *source-to-source*, cujos os ficheiros de entrada são processados e compilados para Java, sendo neste processo criado o *bytecode* que executará na Java Virtual Machine (JVM).

O modelo de execução SOMD é suportado através do sistema de execução X10, fornecendo este as funcionalidades para que métodos Java sejam executados segundo a nossa

especificação em ambientes de memória partilhada e distribuída. O foco desta dissertação encontra-se nos ambientes de memória partilhada, mais concretamente nas arquitecturas de *multi-cores*. O protótipo implementado integra os sistemas de execução da linguagens Java e X10 através da integração de computações em *threads* Java com computações no sistema de execução do X10. Este último é apenas responsável pela execução de métodos SOMD. Os resultados obtidos por este protótipo inicial são bastante promissores, perspectivando assim mais trabalhos futuros nesta solução.

1.3 Contribuições

O objectivo desta dissertação é instanciar o modelo de execução SOMD numa linguagem de programação de grande utilização, nomeadamente o Java. Para tal a API fornecida assumirá a forma de um paradigma *Distribute-Map-Reduce* em que o programador poderá definir (ou controlar) os três estágios.

O protótipo assentará sobre o sistema de execução do X10, partindo de uma base consolidada que nos permitirá concentrar nas funcionalidades que são efectivamente contribuições para o estado da arte.

As contribuições deste trabalho podem então ser resumidas nos seguintes pontos:

1. A apresentação do modelo execução SOMD e do paradigma de programação subjacente *Distribute-Map-Reduce*;

2. O protótipo que instancia este modelo na linguagem Java, usando chamadas ao sistema de execução do X10;

3. Uma avaliação da exequibilidade deste modelo na execução de computações paralelas em máquinas *multi-core*.

1.4 Estrutura do documento

Além deste capítulo de introdução esta dissertação contempla outros seis capítulos estruturados das seguinte forma:

- Capítulo 2 - apresenta-se o levantamento do estado de arte na área da programação concorrente, sendo introduzidos os conceitos mais importantes. Realiza-se também o levantamento das linguagens de programação concorrente, terminando com uma discussão crítica respeitante à escolha do sistema de execução, que dará suporte ao modelo de execução SOMD.

- Capítulo 3 - apresenta-se o modelo de execução SOMD, o seu paradigma de programação e os construtores que darão suporte ao modelo. Ao longo do capítulo serão também apresentados exemplos que ilustram o uso desses construtores.

- Capítulo 4 - define-se a sintaxe concreta da extensão à linguagem Java, apresenta-se o mapeamento dos construtores na linguagem X10, a integração dos sistemas de execução Java e X10, e os passos do processo de compilação que permitem a geração de código Java.
- Capítulo 5 - discute-se os resultados do protótipo relativamente ao seu desempenho, à qualidade da implementação quando comparada com sistema de execução X10, e a nível da produtividade da linguagem.
- Capítulo 6 - apresenta-se o balanço do trabalho realizado face aos objectivos da dissertação, sendo mencionados alguns dos aspectos a abordar como trabalho futuro.

2

Estado da Arte

Neste capítulo serão abordados os temas que enquadram a dissertação a realizar, nomeadamente o estado da arte na área programação paralela. Sendo que se pretende que o protótipo a implementar assente sobre um sistema de execução existente, com suporte para o paralelismo de dados em ambientes de memória partilhada e distribuída, realiza-se um levantamento dos conceitos relativos à programação concorrente e paralela e modelos de programação, fazendo-se ainda referência às linguagens PGAS e APGAS.

O capítulo encontra-se dividido em seis partes: a secção 2.1 relativa a conceitos já consolidados referentes à programação concorrente e paralela, presente nos livros [Bre09, BA06, HS08]; na secção 2.2 são apresentados os modelos de programação paralela mais relevantes actualmente e as dimensões, segundo as quais podemos classificar estes modelos; nas secções 2.3 e 2.4 são apresentados os modelos e linguagens PGAS e APGAS respectivamente; e por fim, na secção 2.5 é elaborada uma discussão crítica às linguagens PGAS e APGAS, de modo a determinar qual deverá ser integrada na implementação.

2.1 Programação Concorrente e Paralela

Nesta secção serão abordados os paradigmas de programação concorrente e paralela, sendo apresentada as suas definições, os seus desafios e as vantagens e desvantagens de cada um dos paradigmas.

2.1.1 Programação Concorrente

O paradigma de programação concorrente consiste na coexistência de vários fluxos de execução (*threads*[1]) de um ou mais processos em execução e ao mesmo tempo, em um ou mais processadores.

Cada *thread* tem acesso exclusivo ao processador por um determinado intervalo de tempo. Findo esse tempo poderá ter de dar o seu lugar a outro e aguardar que chegue novamente a sua vez, verificando-se assim acções intercaladas no tempo por parte destes.

Desafios

O grande desafio deste paradigma é como efectuar a comunicação entre os *threads*, de forma a que estes possam cooperar para levar a cabo os seus objectivos. Esta pode ser realizada via memória partilhada ou através da troca de mensagens. O modelo de memória partilhada consiste na partilha de um espaço de endereçamento por vários *threads* de um ou mais processos, a acederem a um espaço de endereçamento partilhado por todos. A utilização deste modelo implica lidar com problemas de acessos concorrentes, que são normalmente resolvidos recorrendo-se a monitores, *locks* ou semáforos, e ainda lidar com problemas de sincronização que são solucionados através do uso de portas de sincronização para coordenar o trabalho dos *threads*.

O modelo de troca de mensagens consiste na partilha de dados e sincronização de *threads* através do envio e recepção explícita de mensagens entre processos, apresentando cada processo o seu próprio espaço de endereços de memória. As operações de envio e recepção de mensagens são normalmente suportadas por bibliotecas implementadas para o efeito.

Vantagens e Desvantagens

O paradigma de programação concorrente apresenta como vantagens um melhor uso dos recursos do sistema, enquanto um *thread* espera por operações de entrada/saída (I/O), outro pode ser escalonado para fazer uso do processador, permitindo também um maior rendimento (*throughput*), ou seja, um aumento da quantidade de processos concluídos por unidade de tempo.

Em relação às desvantagens, este paradigma implica um controlo de concorrência dos fluxos de execução, que nem sempre é trivial de programar e a possibilidade dos fluxos incorrerem numa situação de *deadlock*, devido à não libertação dos recursos utilizados.

2.1.2 Programação Paralela

A programação paralela trata-se dum subconjunto da programação concorrente, no qual se desenha a aplicação tendo o pressuposto que vários *threads* irão executar efectivamente

[1] A partir deste ponto, por questões de simplificação, assumimos que o *thread* é a unidade básica de concorrência.

em paralelo, sendo cada *thread* atribuído aos processadores disponíveis. O paralelismo pode-se encontrar ao nível da instrução, dos dados ou das tarefas.

O paralelismo permite assim que um programa possa executar mais rapidamente do que se estivesse a correr num só processador (caso o programa esteja preparado para executar em paralelo). Contudo, o ganho que se tem ao aumentar o número de processadores não equivale muitas vezes a um ganho linear no desempenho, devido aos diversos custos existentes (comunicação, processamento de I/O, etc.), ou devido à desadequação do algoritmo paralelo.

Desafios

Paralelizar um programa diz respeito à transformação dum algoritmo sequencial, de modo a prepará-lo para execução em múltiplos processadores simultaneamente, tendo em conta a arquitectura alvo. Para isso muitas vezes é necessário modificar os algoritmos sequenciais utilizados, pois o modelo de programação incute no programador a responsabilidade pela comunicação e pela decomposição do problema, de modo a que este execute eficientemente.

A decomposição do problema refere-se à divisão do mesmo em uma ou mais tarefas que poderão executar em paralelo. Este tema é abordado na secção 2.2, onde serão apresentadas diversas formas de decompor um problema.

Vantagens e desvantagens

Para além das vantagens e desvantagens da programação concorrente, este paradigma tem como vantagem permitir um *throughput* mais elevado, se o problema estiver preparado para executar em paralelo, e como desvantagem incutir no programador a responsabilidade pela comunicação e decomposição do problema, sendo um problema não trivial de se resolver quando se quer o melhor desempenho possível.

Este paradigma encontra-se também muito ligado à arquitectura alvo para qual o problema deve ser decomposto. Isto traz como vantagem, a optimização do problema para essa arquitectura, e como desvantagem, a dificuldade a sua portabilidade para outras arquitecturas.

2.1.3 Lei de Amdahl

A medida de desempenho de um programa paralelo é normalmente calculada através da fracção entre os tempos da versão sequencial do programa e da versão paralela do mesmo, e intitula-se *speedup*.

A lei de Amdahl [Amd67] indica que o *speedup* máximo que se pode alcançar, sem ter em conta factores adversos (e.g. tempo de comunicação), encontra-se limitado pela fracção de código não paralelizável, querendo isto dizer que o aumento do número de processadores a partir de certa altura não apresenta impacto no desempenho do programa. Contudo constatou-se que para um número elevado de aplicações o valor dessa fracção

não é constante, sendo inversamente proporcional ao volume de dados. O aumento do volume de dados das aplicações permite assim obter *speedups* lineares, proporcionais ao número de processadores. Este resultado justifica assim a exploração do paralelismo nos programas.

2.1.4 Taxonomia de Flynn

A arquitectura alvo da execução de um programa paralelo influencia bastante o modo como este irá ser programado. Michael J. Flynn definiu quatro tipo arquitecturas [Fly72] de computadores, que são classificadas sobre duas dimensões, número de fluxos de execução e número de fluxos de acesso a dados:

***SISD* - Single Instruction, Single Data:** Existe apenas um fluxo de execução e que trabalha sobre um conjunto de dados - a execução é sequencial, não explorando qualquer paralelismo. Nesta categoria inserem-se os *mainframes* e o modelo básico da máquina de Von Neumann, a base dos processadores *single-core* e de um *core* num processador *multi-core*.

***SIMD* - Single Instruction, Multiple Data:** A mesma instrução é aplicada simultaneamente a conjunto de dados diferentes, existindo assim paralelismo no conjunto de dados. Máquinas vectoriais, GPGPUs [THO02] e os conjuntos de instruções vectoriais - por exemplo as instruções SSE (*Streaming SIMD Extensions*) [Sie10] - existentes em alguns processadores, são exemplos de implementações deste tipo de arquitectura.

***MISD* - Multiple Instruction, Single Data:** O mesmo conjunto de dados é aplicado a fluxos de execução diferentes em simultâneo. Não se conhecem arquitecturas que se insiram nesta categoria.

***MIMD* - Multiple Instruction, Multiple Data:** Vários fluxos de execução operam sobre vários conjuntos de dados em simultâneo. Esta categoria pode ser decomposta em duas subcategorias: arquitecturas de memória partilhada e de memória distribuída.

Uma arquitectura de memória partilhada consiste em máquinas com vários processadores que partilham a mesma memória física. Mais uma vez, estas podem ser decompostas nas arquitecturas SMP (*Symmetric Multiprocessors*), em que os vários processadores ou *cores* estão ligados a um *bus* partilhado, e as NUMA, em que cada processador tem a sua própria memória privada, apesar de poder aceder à memória dos restantes. Na última o tempo de acesso de um *thread* à memória depende da localidade da mesma, ou seja é rápido se o *thread* aceder à memória local do processador onde está a executar, porém apresenta uma velocidade menor quando acede à memória de outro processador.

As arquitecturas de memória distribuída correspondem a arquitecturas em que vários nós estão ligados por uma rede, sendo que cada nó pode conter vários processadores e a sua própria memória. Exemplos desta arquitectura são o Beowulf [SBS+95], um *cluster* de computadores que está ligado por uma rede local de TCP/IP, os MPPs (*Massively Parallel Processors*) [Bat80] e arquitecturas de sistemas distribuídos.

O trabalho a realizar no contexto desta dissertação foca-se essencialmente em arquitecturas MIMD de memória partilhada, porém o desenho da arquitectura e o protótipo a implementar suportarão também arquitecturas de memória distribuída.

2.2 Modelos de Programação Paralela

Um modelo de programação paralela é um conjunto de conceitos que permite exprimir o paralelismo em programas, podendo a sua implementação assumir a forma de uma linguagem (ou extensão de uma linguagem) de programação ou de um biblioteca. Os modelos podem ser caracterizados segundo várias dimensões, sendo abordadas nesta secção as seguintes: comunicação, decomposição do problema e modelos de execução.

2.2.1 Comunicação

A dimensão da comunicação é o que permite ao programa progredir e atingir o resultado final do problema. Esta pode ser conseguida através de duas formas diferentes, memória partilhada ou troca de mensagens.

Memória Partilhada: Tal como foi referido na subsecção 2.1.1, o modelo de memória partilhada baseia-se na existência de um espaço de memória partilhado, a que vários *threads* acedem para poderem comunicar entre si, sendo este modelo de programação baseado em *threads*, o modelo de base para a programação em memória partilhada. Existem várias APIs que fornecem suporte para este modelo, entre as quais a POSIX Threads [Mue93], também conhecida por Pthreads, e o OpenMP [DM98].

As Pthreads são um padrão POSIX, que especifica uma API para suportar aplicações *multi-threaded*. Essa API fornece funcionalidades de criação e gestão de *threads*, permitindo também o uso de *mutexes*[2], variáveis condição e sincronização de *threads*, através do uso de portas de sincronização. Uma das razões da utilização de *threads* é o facto do tempo de mudança de contexto entre *threads* dum mesmo processo, ser mais rápido do que o tempo de mudança de contexto entre processos. Isto sucede-se devido à partilha de código entre os *threads*, levar a que apenas os registos e o *program counter* dos *threads* sejam trocados aquando da mudança de contexto, sendo esta uma operação muito mais barata do que a mudança de contexto entre processos.

O OpenMP consiste num conjunto de funções de biblioteca e directivas dadas ao compilador, que permitem ao programador inserir paralelismo no seus programas sem ter que se preocupar com a criação e gestão de *threads*. A programação em OpenMP consiste em inserir directivas do pré-processador (*pragmas*), que anotam as zonas do programas em que o normal comportamento de execução deve ser alterado. Esta alteração inicia a execução paralela do programa, não necessitando em muitos casos de modificações a nível do código do programa. O modelo de execução do OpenMP segue o modelo *fork-join*

[2]Permite o acesso em exclusão mútua a zonas de memória

descrito na subsecção 2.2.3.

Troca de Mensagens: O modelo de programação baseado em troca de mensagens tem como base o uso de *sockets* suportados pelo sistema de operação, através da sua interface de chamadas ao sistema. Um *socket* cria um canal de comunicação fiável ponto-a-ponto entre dois processos, sendo a comunicação realizada através de escritas e leituras nesse canal.

O modelo de programação baseado em troca de mensagens está presente em diferentes cenários, por exemplo na programação paralela, com o PVM (*Parallel Virtual Machine*) [Sun90], uma biblioteca que permite a programação de uma rede de computadores como se tratasse de uma máquina virtual, sendo a comunicação feita à base de mensagens. Na programação distribuída em Java com o RMI (*Remote Method Invocation*) [Ora11], uma biblioteca que permite a invocação de métodos em outros espaços de endereçamento, no estilo RPC para aplicações Java; e na programação Web com o protocolo de invocação SOAP (*Simple Object Access Protocol*) [W3C07], que permite a troca de informação estruturada na implementação de *Web Services*.

No âmbito da área que nos interessa o destaque vai para o MPI (*Message Passage Interface*) [SOHL+98], uma biblioteca que é considerada o padrão deste modelo. O MPI especifica funções de comunicação de mensagens entre *threads*. Permite a troca de mensagens síncronas e assíncronas, assim como comunicação ponto-a-ponto e multiponto. Devido ao elevado número de arquitecturas que dispõem de uma implementação desta biblioteca, o MPI é considerado portável.

A segunda versão do *standard* MPI, conhecido por MPI-2 [GLT99], trouxe novas funcionalidades ao *standard* MPI: comunicação *one-sided*, permitindo com operações como *put* e *get*, escrever e ler duma memória remota respectivamente; gestão dinâmica de processos e operações de I/O paralelas.

O MPI deixa a cargo do programador, a responsabilidade pela comunicação do programa e pela decomposição do problema. A paralelização dum programa usando o MPI, tende a ser não trivial, sendo o código gerado bastante mais extenso que o programa sequencial[HCS+05].

Memória Virtualmente Partilhada: Consiste na utilização do modelo de memória partilhada em arquitecturas de memória distribuída, através duma camada de *software* que simula este comportamento. Este modelo de programação pode ser extendido a múltiplas máquinas, tratando-se neste caso de memória partilhada distribuída, sendo este muitas vezes dependente da arquitectura alvo. Como exemplos de *software* que permitem este fim são o GASNet (*Global Address Space Networking*) [BBNY06], os espaços de memória distribuída (*Distributed Shared Memory* - DSM), p.e. TreadMarks [KCDZ94] ou o Kerrighed [MLV+03], linguagens de espaços de endereços globais particionado - PGAS e APGAS (*Asynchronous Partitioned Global Address Space*) - e os espaços de tuplos distribuídos, p.e. modelo Linda [Gel85].

O modelo de memória distribuída partilhada expõe o problema da não uniformização do acesso à memória. Isto acontece devido à memória não se apresentar à mesma distância de todos os processadores, podendo assim provocar deteriorações no desempenho. Este problema é tratado nos modelos PGAS e APGAS com a noção de *localidade*.

2.2.2 Decomposição

A decomposição dum problema diz respeito à forma de reduzir um problema em várias partes tendo em conta a arquitectura alvo, ou seja trata-se da divisão de um problema em problemas mais pequenos, de forma a poderem ser executados em paralelo. Existem duas formas de se obter tal efeito, através do paralelismo nos dados ou do paralelismo nas tarefas.

Paralelismo nos Dados: O paralelismo ao nível dos dados consiste na subdivisão dos dados em conjuntos mais pequenos, que são enviados a *threads* que executam a mesma sequência de instruções sobre cada subconjunto dos dados. Na execução de programas que executem segundo este modelo, pode-se encontrar operações de sincronização de *threads*, para sincronizar as operações a realizar por estes. A figura 2.1 exemplifica o modelo de execução deste modelo de programação.

Este modelo de paralelismo está muito ligado à instrução **forall**[DBK+96], uma variante da instrução for, que permite paralelizar o bloco de instruções delimitado. A paralelização do for diz respeito à divisão de iterações pelos vários *threads* disponíveis, sendo estas independentes entre si.

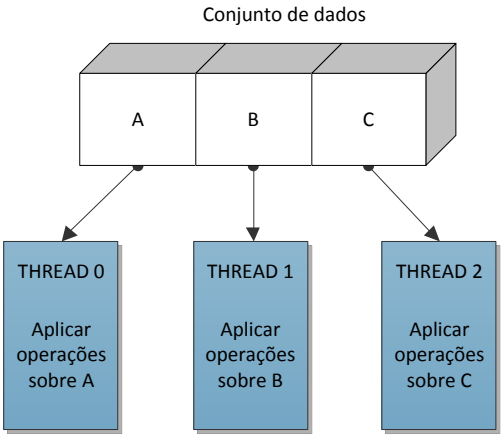


Figura 2.1: Paralelismo de dados

Uma das primeiras linguagens a surgir, que implementa o modelo de paralelismo nos dados, foi o High Performance Fortran (HPF) [MH95]. Esta linguagem é uma extensão à linguagem Fortran, que adiciona o paradigma de programação paralela baseada no modelo de paralelismo de dados, através de directivas passadas ao compilador. Das características mais importantes desta linguagem, destacam-se a distribuição dos dados

sob arquitecturas de memória distribuída, através do uso de *arrays* distribuídos presente no HPF e a sua portabilidade sob as diversas arquitecturas, devido ao uso das directivas passadas sob a forma de comentários, que só são reconhecidas pelo compilador HPF. Esta linguagem liberta o programador das preocupações com os detalhes de comunicação, e da computação paralela dos dados, estando estas dificuldades a cargo do compilador.

A ZPL (Z-level Programming Language) [CCL+98, Bra00] é uma linguagem de programação vectorial, ou seja baseia-se em operações sob escalares para a manipulação de vectores, que introduz a noção de **região** para designar um conjunto de índices de vectores. Este conceito é usado para distribuir os dados dos vectores pelos vários processadores, sendo distribuídos consoante a interacção das regiões dos vectores. Apesar da distribuição dos dados pelos diversos processadores, a ZPL mostra uma visão global dos dados ao programador. Esta noção é usada em muitas outras linguagens, como o X10 [CGS+05], Chapel [CCZ07] e Array Building Blocks (ArBB) [GSC+10]. Outra característica desta linguagem é o facto de, tal como o HPF, libertar o programador das complicações de explicitar como ocorre a comunicação no programa, sendo o compilador responsável por identificar e gerar o código necessário para a comunicação que se adequa a cada máquina onde o programa irá executar.

O Sequoia [FKH+06] é uma linguagem de programação que expõe ao programador a hierarquia de memória no modelo de programação, adoptando a estratégia de "dividir para conquistar", que consiste na divisão do trabalho a realizar em unidades mais básicas que podem ser paralelizáveis. A hierarquia de memória é vista como uma árvore, em que os nós representam as tarefas associadas ao nível de memória em que cada nó se encontra. O nó raíz é responsável pela agregação das computações de todos os outros nós, e os nós folha são responsáveis pelas tarefas a executar na sua forma mais básica. Os restantes nós intermédios são responsáveis pela divisão de tarefas do nó em tarefas mais básicas, e reencaminhar o resultado das tarefas dos filhos ao nó que se encontra no nível acima. O Sequoia usa a noção de *task* para definir a computação, incluindo informação também sobre a comunicação e conjuntos de trabalho. Esta noção expressa o paralelismo do Sequoia, permitindo duas variantes na implementação duma *task*, a variante *inner* e a variante *leaf*. A primeira é aplicada aos nós que não são folhas, representando as tarefas que dividem trabalho e devolvem o resultado da computação dos filhos ao nó pai. A variante *leaf* é aplicada aos nós folha, representando o trabalho a executar na sua forma mais básica. A parametrização das *tasks* é da responsabilidade do programador, cabendo a este a definição das diversas instâncias que representam os níveis de memória da máquina, assim como a definição das tarefas executar em cada nível de memória e ainda definir o valor das variáveis *tunable*, variáveis que podem ser ajustadas para cada um dos níveis de memória.

As linguagens PGAS e APGAS são também exemplos de linguagens que suportam paralelismo nos dados. Visto serem as linguagens que mais se aproximam do trabalho a realizar, serão abordadas de forma mais detalhada em secções dedicadas, nomeadamente nas secções 2.3 e 2.4, respectivamente.

Paralelismo nas tarefas: O paralelismo nas tarefas consiste na decomposição dum problema em várias tarefas mais básicas, que podem executar concorrentemente. O programador é responsável pela identificação das tarefas que depois são atribuídas a um conjunto de trabalhadores (normalmente *threads*). Esta atribuição pode ocorrer de forma estática, sendo no início efectuada a atribuição das tarefas a cada *thread*, ou de forma dinâmica, sendo criada uma fila de tarefas, que os *threads* vão consumindo à medida que o seu trabalho for terminando. A figura 2.2 exemplifica o modelo de execução deste modelo de programação.

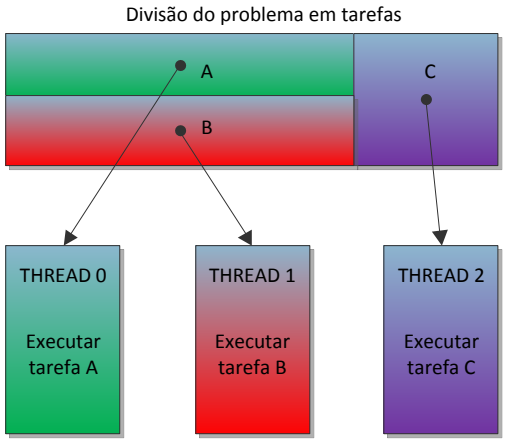


Figura 2.2: Paralelismo de tarefas

Um dos sistemas referência do paralelismo funcional é o Cilk [FLR98], uma extensão à linguagem C, que adiciona funcionalidades de programação paralela. A palavra reservada cilk no cabeçalho das funções, permite que o paralelismo possa ocorrer nessa função quando da invocação desta é precedida pela palavra reservada spawn, ocorrendo assim o paralelismo nas tarefas.

Em relação ao escalonamento das tarefas, cada processador tem uma fila de tarefas prontas a executar e suporta o balanceamento de tarefas entre processadores. O balanceamento das tarefas é suportado, na forma em que processadores sem trabalho vão à procura de trabalhos por executar em processadores mais ocupados, removendo tarefas à cabeça das filas de espera destes. A esta capacidade de retirar trabalho aos processadores mais ocupados deu-se o nome de *work stealing*.

O sistema de execução pSystem [LS97] permite a anotação de programas feitos em C, com poucas modificações à sua sintaxe, de modo a que os programas possam explorar o paralelismo. No pSystem é realiza-se a distinção entre funções C e tarefas. As tarefas são funções C anotadas, que permitem a exploração do paralelismo nessa função, apresentando uma sintaxe um pouco diferente da normal definição de funções C.

Neste sistema de execução é possível definir o número de *workers* (*threads*), parametrizar a heurística a usar pelo escalonador do sistema de execução e a inicialização da memória partilhada. A comunicação efectuada através de memória partilhada, realiza-se

trocando apontadores para zonas de memória partilhada entre tarefas, sendo responsabilidade do programador a distribuição do dados pelas tarefas. Este sistema de execução tem como principal vantagem a parametrização da heurística a usar pelo escalonador por parte do programador, permitindo assim definir funcionalidades como distribuição de carga entre os processadores, ou o roubo de tarefas a outros processadores mais ocupados. Os *workers* comunicam com o escalonador quando precisam de mais tarefas ou existe uma nova distribuição de tarefas.

O di_pSystem [SPL99] extende o pSystem de forma a permitir o seu funcionamento sobre arquitecturas de memória distribuída. Este sistema introduz a noção de **agentes** (processos), responsáveis pela execução em cada máquina do trabalho gerado pelo sistema de execução, dispondo cada agente de um *thread* de comunicação e outro de computação. O *thread* de computação é responsável pela criação, execução e suspensão de tarefas. A computação do trabalho das tarefas pode ser realizado localmente ou remotamente, de modo a evitar *race conditions* entre *threads* e a balancear a carga entre agentes, sendo o escalonador responsável por tomar esta decisão. O *thread* de comunicação é responsável por receber e enviar pedidos de tarefas a outro agentes, comunicando também com escalonador para decidir se a tarefa é realmente enviada a outro agente, ou se é transferida para execução local. A comunicação entre agentes é gerida pelo sistema de execução, passando despercebida ao programador.

A Intel Thread Building Blocks (Intel TBB) [Rei07, CM08] é uma biblioteca de *templates* para a linguagem C++ que permite expressar paralelismo em termos de tarefas, em vez de *threads*. Desta forma o nível de abstracção do programador aumenta, reduzindo o conhecimento que este tem de possuir, para produzir algoritmos paralelos eficientes. As tarefas são mapeadas automaticamente pelo sistema de execução, para *threads* de forma a usar os recursos do processador de maneira mais eficiente. A carga de trabalho é balanceada através do roubo de tarefas a outros processadores. A biblioteca também oferece primitivas para a utilização do paralelismo a nível de dados, permitindo uma maior escalabilidade dos programas, através de funções como parallel_for (baseado no conceito da instrução **forall**) que limita a execução dum ciclo for a um dado intervalo e permite a sua execução em paralelo.

2.2.3 Modelos de execução

Os sistemas de execução paralelos para arquitecturas de memória partilhada normalmente adoptam um modelo *fork-join* (ou *master-slave*), em que as zonas sequenciais do programa são executadas por um único fluxo de execução, enquanto que as zonas susceptíveis de paralelismo são adjudicadas por este mesmo a um conjunto de trabalhadores. No final da execução paralela existe um ponto de sincronização, em que o fluxo principal aguarda pela terminação do trabalho computado em paralelo, para que possa continuar a sua computação. Este modo de execução pode ser repetido várias vezes ao longo da execução do programa. São exemplos desta abordagem são o OpenMP, os executores do

Java e X10 (num só *place*).

Em ambientes de memória distribuída é necessário lançar um processo por máquina. Para tal existem duas abordagens: SPMD e MPMD, sendo que estas usam como modelo base o *fork-join*, porém de formas distintas.

No caso do SPMD, cada fluxo executa o mesmo programa, sendo que nas zonas do código susceptíveis a execução paralela, como por exemplo as delimitadas pela instrução **forall**, cada um opera sobre uma partição dos dados.

Já no caso do MPMD, normalmente um processo assume o papel de mestre e os restantes de trabalhador. O esquema é semelhante ao descrito para memória partilhada - o mestre adjudica trabalho aos trabalhadores disponíveis, e estes poderão realizar tarefas diferentes uns dos outros.

2.2.4 Paralelismos implícito e explícito

Os modelos descritos até este ponto enquadram-se na categoria do paralelismo explícito. Este diz respeito à chamada explícita de funções de biblioteca ou directivas usadas pelo programador para induzir o comportamento concorrente no seu programa, apresentando como vantagem a produção de programas mais eficientes, contudo a sua programação não é trivial.

Por outro lado, o paralelismo implícito diz respeito às transformações automáticas que o compilador, ou o interpretador, de uma dada linguagem, fazem de forma a explorar o paralelismo, sem que o programador se dê conta de tal. O paralelismo é assim criado sem que o programador se aperceba, porém as transformações paralelas geradas geralmente não são as mais eficientes. São exemplos disso algumas implementações do Prolog [CWY91] e do Java [BG97].

2.3 PGAS

O modelo de programação de memória partilhada apresenta ainda um problema que está dependente das arquitecturas dos sistemas, que é o facto de o acesso à memória não ser constante para todos os processadores, pois esta não se encontra sempre à mesma distância. Isto é considerado um problema, pois pode levar a graves problemas de desempenho.

O PGAS (*Partitioned Global Address Space*) resolve este problema expondo ao programador a localização da memória. No PGAS a memória encontra-se dividida em duas partes, uma global e partilhada entre todos os *threads* e outra, privada e local a cada *thread*. A zona de memória global, a todos os *threads*, encontra-se particionada pelo número de *threads*, sendo assim introduzido o conceito de afinidade de memória, também conhecido por localidade. Acessos por um *thread* a qualquer partição, à qual não tenha afinidade, requer comunicação remota. A exposição deste facto ao programador permite que este possa gerir o impacto dos acessos remotos. Tal como outros modelos de programação o

PGAS permite o uso de primitivas de sincronização entre *threads*, como p.e. barreiras. Relativamente à comunicação, no modelo PGAS esta é semelhante à comunicação *one-sided* do MPI 2.0.

De seguida vamos descrever as linguagens Titanium e UPC que instanciam o modelo PGAS. Vamos dar mais destaque à primeira pois no nosso trabalho será realizado no âmbito da linguagem Java.

2.3.1 Titanium

O Titanium [YSP+98] é uma implementação do modelo PGAS para a linguagem Java. É uma extensão à versão 1.4 do Java, que usa as bibliotecas do Java 1.0. O Titanium é uma linguagem orientada a objectos, que faz uso dum *garbage collector* tal como o Java, mas o código é compilado para C e mais tarde para código máquina, não fazendo uso da JVM para executar o código.

Os programas Titanium são executados segundo o modelo de execução SPMD sendo indicado o número de *threads* a executar antes do programa iniciar. Em fase de compilação é indicado qual o tipo de arquitectura onde os programas deverão ser executados, p.e. SMP.

A palavra single pode ser utilizada tanto num método, para indicar que este irá ser executado por todos os *threads*, ou numa variável, para criar uma variável partilhada.

Possui também primitivas de comunicação como: o **barrier**, em que os *threads* se bloqueiam até todos os chegarem à barreira de sincronização; o **broadcast** para enviar um dado valor a todos os outros *threads* a partir dum *thread* específico; o Reduce permite a aplicação de uma função de redução a uma variável presente em todos os *threads*; e o exchange preenche um dado array A, com os valores v específicos a cada *thread*. A listagem 2.1 apresenta algumas das primitivas de comunicação mencionadas num exemplo que tem por objectivo contar o número de ocorrências dum dado número num vector.

Listagem 2.1: Contagem de um dado número num vector

```
1  public single int countNumber(int [1d] array, int num) {
2    single int result = 0;
3    int count = 0;
4    broadcast count from 0;
5    foreach(i in array.domain())
6      if(array[i] == num)
7        count++;
8    Ti.barrier();
9    result = Reduce.add(count);
10   return result;
11 }
```

O conceito de *localidade* no Titanium encontra-se presente na declaração dos apontadores, através do qualificador **local**. Uma variável que não seja qualificada com a palavra **local** é considerada uma variável global, cujo conteúdo pode ser encontrado remotamente. Uma variável que apresente local é considerada uma variável local, cujo conteúdo se encontra localmente. O uso de ambos pode ter impacto significativo na execução dum programa, uma vez que a utilização de apontadores locais incorre num número de instruções menor, quando comparados com os apontadores globais.

No Titanium foi criado um novo tipo de vectores por questões de desempenho, cuja sintaxe difere um pouco da existente para os vectores em Java. Estes novos vectores são construídos a partir de Points, tuplos de inteiros, e RectDomain, um espaço multidimensional constituído por Points. Na criação dum vector é indicado o número de dimensões seguido da letra *d* - p.e. int [3d] cube. É ainda de salientar a presença de açúcar sintáctico sob a forma da construção **foreach** que permite a iteração sobre todos os elementos do vector, sem ter qualquer preocupação relativa a índices. A listagem 2.1 apresenta o uso do **foreach** na linha 5.

Esta linguagem disponibiliza um grande número de operações sobre vectores, como: restrict, que restringe o domínio dum vector; translate, que translada o domínio; e o slice, que usa apenas uma pequena parte, ou "fatia", do domínio; entre outras funções. A listagem 2.2 exemplifica a multiplicação de matrizes usando a operação slice.

Listagem 2.2: Multiplicação de matrizes usando *slices*

```
1 public static void matMul(double [2d] a, double [2d] b, double [2d] c) {
2   foreach (ij in c.domain()) {
3     double [1d] aRowi = a.slice(1, ij[1]);
4     double [1d] bColj = b.slice(2, ij[2]);
5     foreach (k in aRowi.domain()) {
6       c[ij] += aRowi[k] * bColj[k];
7     }
8   }
9 }
```

O Titanium extende a noção de tipos primitivos do Java através do uso de classes imutáveis. Uma classe imutável não é uma subclasse de java.lang.Object, não sendo por isso possível atribuir o valor **null** às variáveis deste tipo de classes. Outra restrição é a existência de um construtor por omissão que não pode apresentar parâmetros. De salientar ainda que estas classes não podem estender outras, nem implementar interfaces. Nestas classes é permitida ainda a redefinição dos operadores. A listagem 2.3 apresenta a definição duma classe imutável para representar um número complexo, tendo sido redefinida a operação de soma para esta classe.

Listagem 2.3: Classe imutável para representar um número complexo

```
1  immutable class Complex {
2    public double real;
3    public double imag;
4
5    Complex () {
6      real=0;
7      imag=0;
8    }
9    public Complex(double r, double i) {
10     real = r; imag = i;
11   }
12   public Complex op+(Complex c) {
13     return new Complex(c.real + real, c.imag + imag);
14   }
15 }
```

2.3.2 UPC

A linguagem UPC (*Unified Parallel C*) é uma extensão à linguagem C, que implementa o modelo PGAS. O UPC utiliza a palavra reservada **shared** para diferenciar os dados partilhados dos privados. Assim uma variável que tenha na sua declaração a palavra **shared**, fica acessível a todos os *threads*. As variáveis pertencentes ao espaço privado de cada *thread* são declaradas da mesma forma que no C. A partilha dos dados dos vectores é feita ao bloco, podendo o programador escolher o tamanho dos blocos a distribuir pelos *threads*. O UPC segue o modelo de execução SPMD, fazendo uso de expressões como o **upc_forall**, que distribui iterações independentes pelos diversos *threads*.

Em relação à sincronização possui barreiras bloqueantes, **upc_barrier**, e barreiras não bloqueantes, conhecidas por *Split-Phase Barriers*, em que são distinguidas as fases de notificação e de espera. A fase de notificação é não bloqueante, permitindo ao *thread* notificar outros *threads* e prosseguir o seu trabalho, é usado através da primitiva **upc_notify**. A fase de espera é bloqueante, tendo o *thread* de esperar que os outros terminem o seu trabalho para poder prosseguir, é usado através da primitiva **upc_wait**. Entre outras formas de sincronização que o UPC permite, destacam-se as *fences*, **upc_fence**, que asseguram que a partir daquele ponto qualquer referência a variáveis partilhadas feita anteriormente está completa, e os *locks*, através do uso das primitivas **upc_lock**, **upc_unlock** e o **upc_lock_attempt**. Esta última devolve o valor 1, se esta operação tiver obtido o *lock* e zero caso contrário.

Esta linguagem permite ainda ao programador a escolha entre dois modelos de consistência de memória, relaxado e estrito. Ao usar o modelo de consistência relaxado, as operações sob variáveis partilhadas podem ser reordenadas, ou mesmo ignoradas pelo compilador, ou pelo sistema de execução, caso não afectem o resultado final, enquanto que no modelo estrito esta situação já não se sucede, a ordenação sequencial das operações sob variáveis partilhadas é garantida, não havendo operações intercaladas no tempo

sob estas variáveis.

2.4 APGAS

O modelo de programação APGAS (*Asynchronous PGAS*) [SAB+10] estende o modelo do PGAS com a faculdade da composição funcional, através dos conceitos de *localidade* e de *actividade*.

O conceito de *place* consiste numa unidade computacional que tem uma colecção finita de *light-weight threads* residentes, denominadas *actividades*, e uma quantidade limitada de memória partilhada, que é acessível de forma uniforme por todos os *threads* do *place*.

Uma *actividade* representa uma actividade computacional realizada numa dada *localidade*, que permite a execução de código assincronamente, podendo ser usada para transferir dados, p.e. copiar um vector duma dada localidade para outra. O lançamento de uma *actividade* pode ser feito localmente ou remotamente, permanecendo esta na *localidade* em que foi lançada durante todo o seu tempo de vida. Aceder a dados remotos numa *actividade* apresenta algumas limitações, sendo tal apenas possível se for lançado um novo *actividade* na localidade de destino. Com o intuito de controlar a execução das *actividades*, foi introduzido o *finish*, que se trata duma porta de sincronização, que permite à actividade pai bloquear, até que todas as actividades filhos terminem.

2.4.1 X10

O X10 é uma linguagem de programação orientada a objectos, inspirada na linguagem Java, e que instancia o modelo de programação APGAS. Nesta linguagem as *localidades* foram denominadas *places* e as *actividades* denominadas *asyncs*.

Os *asyncs* para além de referenciar objectos do seu *place*, podem referenciar objectos em outros *places*, fornecendo assim o conceito de espaço de endereçamento global comum às linguagens PGAS.

Os exemplos de programação nesta linguagem ilustrados ao longo deste capítulo apresentam uma sintaxe ligeiramente diferente da linguagem Java, por isso recomenda-se a leitura da especificação da linguagem [IBM11] para mais informações sobre a mesma.

A palavra reservada **at** determina em que *place* as instruções do bloco delimitado devem ser executadas. Os dados referenciados no bloco do **at** são copiados para o *place* onde irá ocorrer a computação, originando assim uma redundância nos dados, havendo uma cópia local e outra remota no *place* em questão. Esta actividade de mandar executar código noutro *place* é uma actividade síncrona. O número de *places* é determinado no início do programa, não sendo permitida a sua alteração durante a sua execução.

A palavra reservada **async** indica que as instruções presentes no bloco deste, devem ser computadas independentemente do *thread* invocador. O **async** representa o conceito de *actividade* existente no APGAS. As *actividades* podem ser encapsuladas por um bloco

Listagem 2.4: Soma paralela de 2 vectores usando DistArray

```
1  static def sumDistArrays(a1:DistArray[Int], a2:DistArray[Int],
       a3:DistArray[Int]): void {
2    finish for (d in Place.places())  at(d) async {
3      val dist = a1Dist.dist.get(d);
4      for(i in dist) {
5        a3Dist(i) = a1Dist(i) + a2Dist(i);
6      }
7    }
8  }
```

finish,que garante a terminação global da actividade antes que o resto da computação local possa prosseguir.

A linguagem X10 utiliza a estruturas de dados DistArray para suportar *arrays* distribuídos, tendo esta estrutura a capacidade de particionar os dados pelos diversos *places*. Os acessos a dados no próprio *place* através desta estrutura não necessitam de qualquer operação auxiliar, contudo acessos a dados em outros *places* implicam a mudança da execução para o *place* em questão. Por exemplo o acesso a dados localizados num *place* B a partir de um *place* A, implica a mudança da execução do programa para o *place* B, onde aí então se poderá aceder aos dados em questão. A listagem 2.4 ilustra a soma de dois vectores, sendo o resultado guardado num terceiro vector, usando DistArrays. Neste exemplo procede-se à criação de um **async** por cada *place* (linha 2). Cada **async** obtém a partição dos dados referente ao *place* onde executa (linha 3), procedendo de seguida à soma dos vectores. Estas acções encontram-se encapsuladas pelo bloco **finish** de modo a garantir, que quando o método só termina, quando a computação da soma em paralelo assim o faz também.

As variáveis partilhadas no X10 são encapsuladas por instâncias da classe GlobalRef. Esta estrutura tem como objectivo guardar os dados relativos à variável partilhada num dado *place*, guardando informação sobre a variável partilhada e sobre o *place* onde esta está localizada.

Com vista a trabalhar sobre ambientes de programação de memória partilhada e de modo a que o acesso a dados partilhados não produza resultados inconsistentes, o construtor **atomic** aparece como um construtor de alto nível capaz de coordenar o acesso aos dados partilhados, encapsulando as operações a realizar de forma atómica. A listagem 2.5 ilustra um método para contar o número de ocorrências dum dado número num DistArray, recorrendo a uma variável partilhada para guardar o seu resultado. A criação da variável partilhada ocorre na linha 2, correspondendo a linha 3 à atribuição do seu valor inicial. Nas linhas 5 a 14 encontramos o lançamento da execução de *actividade* em cada *place* de modo a calcular o número de ocorrências, nas linhas 9 a 11 é apresentado o código relativo à alteração do valor da variável partilhada, no qual a execução é enviada para o *place* que guarda essa variável, e aí é realizada o incremento da variável dentro dum bloco **atomic**. O comando at(count) corresponde a açúcar sintáctico para mudar a

Listagem 2.5: Contagem do número de ocorrências de um número num DistArray

```
1   public static def countNum(array: DisttArray[Int], num: Int) {
2     val count = GlobalRef[Array[Int](1)](new Array[Int](1));
3     count()(0) = 0;
4
5     finish for (d in Place.places())  at(d) async {
6       val dist = array.dist.get(d);
7       for(i in dist) {
8         if(array(i) == num) {
9           at(count) {
10            atomic count()(0)++;
11          }
12        }
13      }
14    }
15
16    Console.OUT.println("Count" + g1()(0));
17  }
```

execução para o *place* onde está guardada a variável partilhada.

A sincronização de actividades no X10 é baseada no conceito de *clock*, que pode ser visto como uma porta de sincronização com diversas fases. O uso do *clock* garante que todas as *actividades* registadas neste avançam a uma próxima fase, quando todas estas terminarem a computação da fase precedente. Uma *actividade* pode estar registada em mais do que um *clock*. A listagem 2.6 ilustra o caso em que são lançadas duas actividades, cuja computação em cada fase diz respeito à impressão de uma String que contém o número da fase em que se encontra. As fases encontram-se separadas pela instrução cl.advance() que funciona como uma barreira de sincronização, garantindo assim que não haverão operações de fases diferentes intercaladas no tempo.

A linguagem X10 apresenta duas implementações do seu sistema de execução, uma em C++ e outra em Java. A implementação em C++, implica um processo por *place* e uma *thread pool* por processo, não havendo balanceamento de trabalho entre *places*. A implementação em Java conta com uma JVM para cada *place*, uma *pool* de *threads* para cada *place*, e utiliza o conceito de *work stealing* para balancear trabalho pelos *threads*.

2.4.2 Outras linguagens

A linguagem Chapel é outra instância do modelo APGAS, em que o conceito de *place* é denotado como **locale** e o conceito *actividade* como o bloco **begin**, suportando tal como o X10, uma visão global do espaço de endereçamento.

Numa versão inicial o X10 foi implementando como uma extensão à linguagem Java, tendo posteriormente evoluído para uma linguagem independente. Nesse contexto, o Habanero-Java prossegue o trabalho realizado na versão inicial do X10, apresentando-se como uma extensão à linguagem Java. Esta linguagem é usada a nível académico

Listagem 2.6: Uso de *clocks* no X10

```
1  static def say(s:String) = {
2    atomic{x10.io.Console.OUT.println(s);}
3  }
4
5  public static def main(argv:Rail[String]) {
6    finish async{
7      val cl = Clock.make(); // Criacao do clock
8      async clocked(cl) { //inicia uma activity A registada no clock cl
9        say("A-1");
10       cl.advance(); // funcao bloqueante que garante que so avanca para a proxima fase,
11       say("A-2"); // quando todos os clocks registados na actividade mudam de fase
12     }
13     async clocked(cl) { //inicia uma activity B registada no clock cl
14       say("B-1");
15       cl.advance();
16       say("B-2");
17     }
18   }
19 }
```

no ensino da programação paralela aos estudantes e a nível de investigação, como linguagem de testes. Não suporta genéricos nem anotações e tem suporte para vectores multi-dimensionais, que são definidos por dois pontos, um limite superior e um limite inferior, que definem *regiões*. Suporta também números complexos como tipo primitivo da linguagem.

O Habanero-Java herda a maior parte dos conceitos do X10, nomeadamente os *places*, os blocos *async* e *finish*. As diferenças centram-se na utilização explícita de futuros (algo que foi abandonado pelo X10) e a sincronização de actividades baseadas em *phasers*. Um *phaser* assemelha-se ao *clock* existentes no X10, sendo uma primitiva de sincronização. Existe ainda o *foreach* que permite a iteração paralela dos pontos dos vectores *multi-dimensionais* duma dada região, sendo cada iteração lançada em paralelo.

2.5 Discussão Crítica

Finda a apresentação dos fundamentos da programação paralela e de algumas das linguagens mais próximas com o nosso trabalho, culminamos este capítulo com uma pequena discussão critica. Esta foca, por um lado, o estado da arte no que se refere ao suporte linguístico para o paralelismo de dados, e por outro, que sistema de execução é o mais apropriado para o desenvolvimento do nosso trabalho.

O paralelismo de dados apresenta-se na sua essência ao nível dos ciclos do programa. Instruções como o **forall** definem o paralelismo a este nível, onde o trabalho dos ciclos é dividido pelos vários trabalhadores existentes. Os construtores a introduzir na linguagem devem então apresentar este comportamento permitindo dividir o trabalho da iteração dos ciclos pelos trabalhadores.

Ao longo deste capítulo foram apresentados os conceitos e linguagens de programação relacionados com a programação A nível da escolha da arquitectura alvo o nosso foco vai para arquitecturas MIMD de memória partilhada, mais propriamente sistemas *multi-core*, porém na implementação do protótipo este suportará sistemas de memória partilhada distribuída.

No que se refere ao suporte à execução do modelo SOMD em Java, os nossos requisitos eram 1) que a linguagem fosse uma extensão ao Java ou fosse compilada para Java e 2) que suportasse a distribuição de dados em arquitecturas de memória partilhada e distribuída. Neste contexto, as opções possíveis são Titanium, X10 e o Habanero-Java. Destas a nossa escolha recai sobre o X10, sendo explicado o porquê dessa decisão nos parágrafos seguintes.

O Titanium tem como vantagens a fácil integração de classes Java com código Titanium, podendo as classes Java chamarem métodos de classes Titanium sem grande dificuldade, sendo a sintaxe do Titanium bastante semelhante à do Java. Apresenta também de forma explícita o conceito de *localidade* existente no modelo PGAS. Em relação às desvantagens salientamos o facto de o Titanium não correr numa JVM, sendo o código compilado para C. As classes Java que queiram usar classes Titanium têm de ser compiladas com o compilador de Titanium, o que inviabiliza o uso de bibliotecas disponíveis para o desenvolvimento de aplicações Java. Além de mais, as bibliotecas *standard* do Java que o Titanium dispõe encontram-se bastante desactualizadas (versão 1.0). Estas não incluem a serialização de objectos Java, o que inviabiliza o envio de objectos através de canais de comunicação (como um socket).

O X10 que instancia o modelo APGAS, apresenta mais funcionalidades do que o Titanium, contudo estamos apenas interessados nas funcionalidades relativas ao paralelismo de dados. O X10 pode ser compilado para Java, sendo a sua execução suportada pelo sistema de execução X10JRT cujo código e documentação se encontra disponível na página da linguagem. Esta linguagem ao contrário do Titanium suporta a serialização de objectos, o que permite a execução de programas em ambientes de arquitectura de memória partilhada distribuída. Fornece uma biblioteca também mais rica que o Titanium, p.e. DistArray e os GlobalRef. Como desvantagens o X10 apresenta uma sintaxe diferente da do Java, o que para criar programas de exemplo poderá dificultar numa fase inicial, tendo de para isso de se recorrer à especificação da linguagem.

A alternativa Habanero-Java tem a vantagem de ser uma extensão ao Java, o que facilita na criação de programas nesta linguagem e na integração do sistema de execução desta linguagem com o código Java dos programas SOMD. No entanto, esta linguagem aparece ligada ao ensino da programação paralela o que pode ser uma desvantagem, por não apresentar mais análises comparativas de desempenho com outras linguagens existentes (como por exemplo o próprio X10 que deu origem à linguagem).

Podemos então concluir que a linguagem que apresenta mais vantagens para ser integrada na extensão à linguagem Java é o X10. Este comparativamente com o Titanium tem

a vantagem de conseguir ser executado numa JVM, sendo por isso mais fácil a sua integração com o sistema de execução Java. O X10 também ganha vantagem em relação ao Habanero-Java, pois encontra-se numa versão mais maturada que o Habanero-Java (que surgiu a partir da versão 1.5 do X10) tendo já sofrido profundas alterações e evolução do seu desempenho.

3

Modelo de execução SOMD

Neste capítulo é descrito o modelo de execução SOMD, apresentado o modelo de programação inerente, definidos os construtores que dão suporte ao modelo e ilustrados alguns exemplos de programação usando estes construtores.

3.1 O Modelo de Execução

O modelo de execução SOMD consiste na execução de várias instâncias duma dada sub-rotina em paralelo sobre partições distintas dos dados de entrada, ou seja trata-se da transposição do conceito de SPMD para a execução de uma sub-rotina, seja ela uma função, um método ou uma operação de computação orientada a serviços.

Este modelo é transparente para o invocador, desacoplando a invocação da execução, sendo semelhante ao que pode ser encontrado nos modelos de actor [Agh86] e objecto activos [LS95]. A figura 3.1 ilustra este comportamento, a invocação é realizada de forma síncrona e a execução realizada por múltiplos fluxos de execução. Nesta imagem é ainda possível identificar duas fases importantes deste modelo, a fase de distribuição de dados e a de redução dos resultados, que serão abordadas na secção 3.2.

Com este modelo pretendemos fornecer ao programador o paralelismo de dados nas suas aplicações, sem se ter preocupar com a gestão e criação dos *threads* e com um conjunto mínimo de alterações no código, e.g. o uso das palavras reservadas será o suficiente desencadear a paralelização do programa, disponibilizando assim o modelo de paralelismo de dados aos programadores sem experiência na área.

Este modelo pode ser aplicado tanto a ambientes locais - e.g. *multi-cores* - como a ambientes distribuídos, sendo muitas vezes a implementação da sub-rotina, que use este

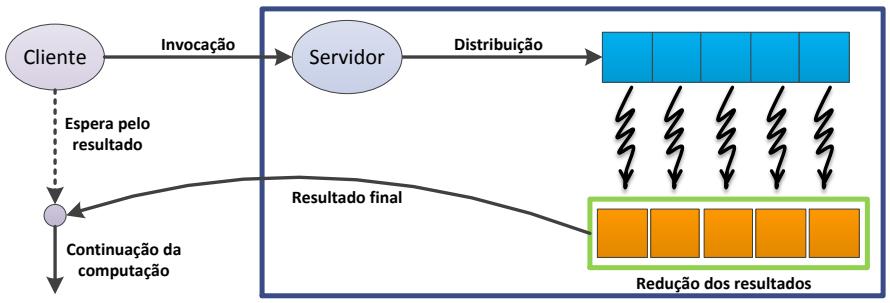


Figura 3.1: Transparência para o invocador (retirado de [MP12])

modelo de execução, indiferente à arquitectura em que irá ser executado. As diferenças de implementação dependentes da arquitectura encontram-se ao nível das variáveis partilhadas. Em ambientes de programação *multi-core* os parâmetros do método, que não estão sujeitos a uma distribuição, são partilhados por todos os *threads* e no âmbito de ambientes distribuídos tal requer uma camada de *software* que ofereça o suporte de memória partilhada sobre ambientes de memória distribuída, como o GASNet. A figura 3.2 mostra a execução do modelo num ambiente distribuído, como é o caso dum *cluster*. Neste, a execução segue o modelo mestre-trabalhador, onde o nó que responde à invocação do método assume o papel de mestre, sendo responsável por realizar a distribuição dos dados, lançar a computação para os nós trabalhadores (executando ele próprio parte da computação); colectar os resultados parciais, e; aplicar a função de redução a esses resultados. A distribuição dos dados de entrada num *cluster* é realizada a dois níveis, ao nível dos nós alvo e ao nível dos trabalhadores existentes em cada nó. No que diz respeito à redução, esta é realizada apenas no nó mestre devido a: 1) que normalmente os resultados parciais devido à sua dimensão, não justificam a sua aplicação paralela, e 2) a aplicação hierárquica da política de redução pode comprometer a correcção dos resultados computados. Por exemplo, dado o conjunto de dados $v_i : i \in [1, n+m]$:

$$\text{média}(\text{média}(v_1, ..., v_n), \text{média}(v_{n+1}, ..., v_{n+m})) \neq \text{média}(v_1, ..., v_n, v_{n+1}, ..., v_{n+m})$$

Por isso, invés de se expor estes detalhes ao programador, optou-se por relegar estes para a implementação da estratégia de redução, que por sua vez poderá recorrer a um método SOMD para computar o resultado.

3.2 Modelo de Programação - Paradigma Distribute-Map-Reduce

O modelo de execução é apresentado ao programador como um paradigma *Distribute-Map-Reduce* (DMR) referente às várias fases de execução da sub-rotina, nas quais podemos encontrar as fases de:

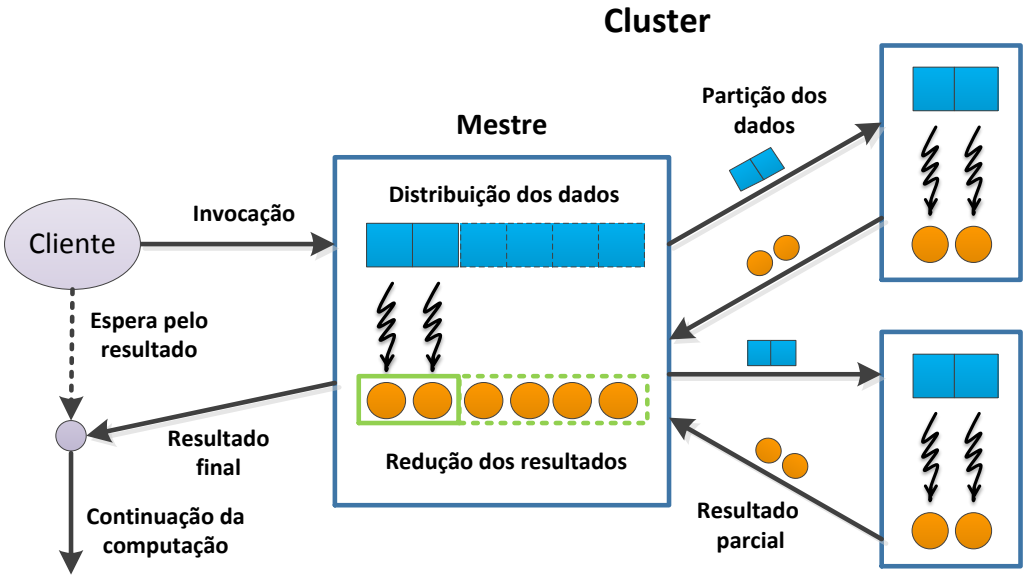


Figura 3.2: Execução em ambiente distribuído (retirado de [MP12])

distribuição de dados - corresponde ao particionamento dos parâmetros da sub-rotina em múltiplos subconjuntos que serão atribuídos aos *threads* que irão executar o método. A criação das partições consiste na criação de cópias dos dados consoante a distribuição definida. As distribuições podem ser aplicadas a parâmetros do método ou a variáveis locais, podendo ser efectuadas mais do que uma distribuição nesta fase, desde que sejam aplicadas a parâmetros diferentes;

aplicação da sub-rotina - aplicação em paralelo de uma instância do método sub-rotina sobre cada partição dos parâmetros de entrada. Cada execução produz um resultado que é passado para o estágio seguinte;

redução dos resultados - consiste na aplicação duma função de redução ao conjunto de resultados computados pela fase anterior, de forma a obter apenas um único resultado final (o resultado da sub-rotina). É apenas permitido aplicar uma redução para cada método.

A figura 3.3 é representativa do modelo DMR, sendo possível identificar as fases de distribuição, execução e de redução. Na fase de distribuição é apresentada de forma simplificada na medida em que é apresentada apenas uma distribuição, mas podem ocorrer várias distribuições sobre dados diferentes. Dado um argumento do tipo T, a distribuição sobre tal argumento deve ser uma função do tipo

$$T \mapsto Vector < T >$$

Em relação à fase de redução, esta devolve um valor do tipo R, devendo representar uma função do tipo

$$Vector < R > \mapsto R$$

O facto de se usar vectores na representação destas funções, prende-se com o facto de ser necessário manter a informação sobre a ordem das partições e dos resultados parciais, de modo a computar um resultado correcto. Por exemplo o elemento i do vector dado à redução é o resultado da aplicação da sub-rotina à partição i da distribuição do conjunto de dados inicial.

Deste modo, cada instância do método (uma vez que nesta dissertação como se aplicou o modelo SOMD a uma linguagem orientada a objectos, a partir deste ponto em diante usar-se-á a terminologia método invés de sub-rotina) está de acordo com o seu protótipo original, uma vez que recebe um dos elementos da distribuição (do tipo T) e produz um resultado do tipo R.

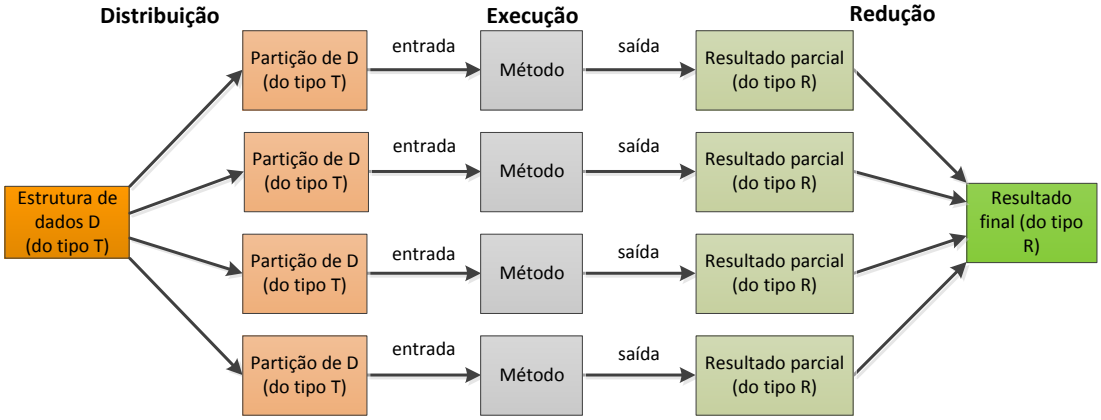


Figura 3.3: Forma básica do modelo de execução (retirado de [MP12])

O objectivo deste trabalho é fornecer mecanismos que permitam ao programador comum poder tirar partido da natureza paralela do *hardware* alvo com um esforço mínimo. Para tal antevê-se que seja disponibilizadas sob a forma de biblioteca o conjunto das distribuições e reduções mais utilizadas.

Neste contexto, o programador terá apenas de escolher as políticas de distribuição e redução a aplicar, sem que para isso tenha de modificar o código do método alvo.

No entanto, tal não impede o programador de definir a sua própria política de distribuição, ou redução, ou definir uma implementação do método, ciente de que este vai ser executado segundo o modelo SOMD, quer questões de algoritmia, quer por questões de desempenho.

3.3 Exemplos de programação

O modelo apresentado foi instanciado numa extensão à linguagem Java, fazendo uso de todos os construtores disponíveis na mesma, mais os construtores que servirão de suporte à execução do modelo.

Optámos por adicionar construtores à linguagem em detrimento da definição de um conjunto de anotações, devido ao facto deste trabalho estar inserido num projecto mais vasto que visa adicionar o paradigma de paralelismo de dados a um modelo de programação baseado em serviços, estando este a ser instanciado como uma extensão à linguagem Java. Apesar do modelo SOMD ser ortogonal ao modelo dos serviços, optou-se pela mesma solução para manter a coerência.

Começamos por apresentar os construtores base para a definição das políticas de distribuição e redução: o **dist** e o **reducewith**, sendo apresentada a sua descrição e a sua notação de modo a se perceber os exemplos de programação. Mais detalhes sobre a integração da linguagem Java com estes construtores serão apresentados na secção 4.2.

dist - define a política de particionamento de um dado parâmetro de entrada (ou variável local, como será descrito em 4.2). A sintaxe definida para os construtor é a seguinte:

$$\textbf{dist}\ C(\vec{e})\ \textit{declaração\_de\_variável}$$

onde $C(\vec{e})$ denota o construtor duma classe que implementa a política de distribuição. No caso em que o tipo da estrutura alvo é um vector, a classe que determina a política de distribuição pode ser omitida em detrimento de uma política por omissão. Esta particiona o vector da forma mais equitativa quando possível pelos trabalhadores existentes.

reducewith - define a política de redução a ser aplicada aos resultados parciais provenientes das múltiplas execuções do método. A sintaxe da aplicação da política ao método é a seguinte:

$$\textit{cabeçalho\_do\_método}\ \textbf{reducewith}\ C(\vec{e})\ \{\textit{corpo\_do\_método}\}$$

onde $C(\vec{e})$ é o construtor duma classe que implementa a política de redução.

Tanto as classes de distribuição, assim como as classes de redução, têm de seguir as especificações das interfaces Distribution e Reduction respectivamente, sendo estas apresentadas na subsecção 4.4.

De seguida apresentam-se alguns exemplos ilustrativos da aplicação de ambos os construtores. A listagem 3.1 apresenta um método que soma o conteúdo de dois vectores, sendo ambos anotados para serem distribuídos de acordo com a política por omissão.

A redução apresentada na listagem 3.2, ArrayAssembler, cria um vector a partir dos vectores parciais gerados pela execução de cada instância do método. A redução recebe um parâmetro que indica o tamanho do vector a construir, tal pode ser omitido à custa da penalização do desempenho, o que implica a descoberta desse mesmo tamanho a partir dos resultados parciais.

Em relação à redução ainda mais uma nota referente ao construtor da classe da redução em causa, que recebe como argumento o tamanho do vector a criar.

É ainda de salientar que o valor da expressão array1.length toma valores diferentes em partes distintas do código. Quando é passado ao construtor da redução este tem o valor do tamanho do vector sem ser particionado, e quando é passado como argumento para criar um novo vector no código, o seu valor é do tamanho do vector particionado durante a fase de distribuição.

Listagem 3.1: Função que soma dois vectores em paralelo

```
1  public int[] sum(dist int[] array1, dist int[] array2) reducewith
       ArrayAssembler(array1.length) {
2    int[] array = new int[array1.length];
3    for(int i = 0;  i < array.length; i++) {
4      array[i] = array1[i] + array2[i];
5    }
6    return array;
7  }
```

Listagem 3.2: Redução ArrayAssembler

```
1  public class ArrayAssembler<T> implements Reduction<T[]> {
2
3    private int size;
4
5    public ArrayAssembler(int size) {
6      this.size = size;
7    }
8
9    public int[] reduce(T[][] array) {
10     T[] result = new T[size];
11     int count = 0;
12     for(int i = 0; i < array.length; i++)
13       for(int j = 0; j < array[i].length; j++) {
14         result[count++] = array[i][j];
15       }
16     return result;
17   }
18 }
```

A listagem 3.3 ilustra a definição dum método que calcula o número de nós existente numa dada árvore. Sendo a estrutura de dados a usar uma árvore e não um vector, a

distribuição por omissão não é aplicável. Assim sendo, é indicada a classe que implementa uma função de distribuição sob árvores, TreeDist, que particiona a árvore original em várias sub-árvores, como ilustrado na listagem 3.4.

Este exemplo apresenta ainda uma pequena particularidade, a decomposição de método original em dois. Tal acontece porque o método original é recursivo e a sua aplicação obriga à criação de um método auxiliar que realize. Isto acontece porque caso aplicássemos apenas o método countSizeParallel, este iria ser chamado recursivamente para cada partição de dados, aplicando assim paradigma DMR sucessivamente, até já não haver mais dados para particionar. Isto provocaria assim um custo no desempenho que penalizaria a execução do programa. Assim recorreu-se à função auxiliar countSize para evitar que tal aconteça.

A política de redução SumReduce, apresentada na listagem 3.5, soma todos os resultados parciais, de modo obter o número total de nós da árvore.

Listagem 3.3: Contagem do número de nós duma árvore Tree

```
1  public int countSizeParallel(dist:TreeDist() Tree t) reducewith
       SumReduce() {
2    return countSize(t);
3  }
4
5  public int countSize(Tree t) {
6    int sum = 0;
7    for(Tree s : (List<Tree>)t.sons)
8      sum += countSize(s);
9    return 1 + sum;
10 }
```

Listagem 3.4: Distribuição TreeDist

```
1  public class TreeDist<A> implements Distribution<Tree<A>> {
2      public Tree<A>[] distribute(Tree<A> tree, int n) {
3        ArrayList<Tree<A>> a = new ArrayList<Tree<A>>();
4        ArrayList<Tree<A>> b = new ArrayList<Tree<A>>();
5        Tree<A> nil = new Nil<A>() ;
6        a.add(tree) ;
7        for (int i = 0 ; i < n ; i++ ) {
8          ArrayList<Tree<A>> u = a; a = b; b = u;
9        a.clear() ;
10         for ( Tree<A> t : b ) {
11           a.add(t.IsEmpty() ? nil : t.Left()) ;
12           a.add(t.IsEmpty() ? nil : t.Right()) ;
13         }
14       }
15       a.add(0, tree.Copy(n)) ;
16       return a.toArray();
17     }
18 }
```

Listagem 3.5: Redução SumReduce

```
1  public class SumReduce implements Reduction<Integer> {
2    public Integer reduce(Integer[] array) {
3      int result = 0;
4      for(int i = 0; i < array.length; i++){
5        result += array[i];
6      }
7      return result;
8    }
9  }
```

3.3.1 Variáveis partilhadas e sincronização

Até esta altura apenas os parâmetros passados aos métodos podem ser partilhados pelas várias instâncias em execução (relembra-se que tal não inclui os parâmetros distribuídos). Portanto, a menos destes parâmetros, o corpo de cada método define um contexto de execução isolado dos restantes. No entanto, alguns problemas beneficiam da partilha de estado entre as instâncias do métodos. Com esse pressuposto estendemos o modelo base com o construtor shared.

Porém o uso de variáveis partilhadas é gerador de alguns problemas de concorrência, nomeadamente problemas de sincronização e de acesso concorrente ao estado partilhado das variáveis. Para lidar com estes dois problemas foram adicionados os construtores **barrier** e **atomic**.

barrier - indica um ponto de sincronização no código, onde todos as instâncias dos métodos (IM) devem aguardar até, que estes se encontrem todos nesse ponto do código. Este construtor está ao nível dos comandos e a sua sintaxe é:

barrier

atomic - para sincronizar o acesso às variáveis partilhadas é permitida a definição de blocos atómicos, ou seja o conjunto de operações será visto como uma só operação e apenas o resultado final das variáveis será visto pelas IMs. Este mecanismo difere do existente no Java, em que os blocos **synchronized** adquirem o monitor do objecto, na medida em que sincroniza apenas as instâncias dos métodos em execução. Este construtor é usado da seguinte maneira:

atomic$\{comandos\}$

Estes construtores fornecem ao programador mecanismos para que a execução dos métodos seja determinística, não estando sujeita ao escalonamento dos *threads* e aplicar este modelo a uma gama maior de problemas.

A listagem 3.6 faz uso dos construtores mencionados acima. Neste exemplo pretende-se normalizar os valores presentes num vector, através da subtracção do valor mínimo

existente neste. Na linha 2 é declarada uma variável local partilhada, nas linhas 7 a 9 é modificado o valor da variável em exclusão mútua, e na linha 10 procede-se à sincronização das IMs de modo a garantir que todos procedem à normalização dos resultados, apenas quando todos se encontrarem na **barrier**.

Listagem 3.6: Variáveis partilhadas e construtores de sincronização

```
1  public int[] normalize(dist int[] array) reducewith
       ArrayAssembler<Integer> {
2      shared int globalMin = Interger.MAX_VALUE;
3      int localMin = array[0];
4      int[] result = new int[array.length];
5      for (int i = 1; i < array.length; i++)
6        if (array[i] < localMin)
7          localMin = array[i];
8      atomic {
9        if (localMin < globalMin)
10         globalMin = localMin;
11     }
12     barrier;
13     for (int i = 0; i < result.length; i++)
14       result[i] = array[i] - globalMin;
15     return result;
16 }
```

3.3.2 Construtor *distshared*

Alguns problemas beneficiam da sobreposição de partições, nomeadamente para permitir leituras e escritas a posições para além dos limites definidos pela distribuição. Quando se trata de apenas de operações de leitura, tal pode ser conseguido com a criação duma estratégia de distribuição, que realize o particionamento dessa forma. Porém uma vez que valores distribuídos não são partilhados pelas instâncias dos métodos, modificações a estes por parte dum *thread* não são propagadas para as restantes. Para superar a esta limitação propomos a noção de vector partilhado distribuído, **distshared**. Este construtor assemelha-se ao construtor **dist**, realizando à mesma a partição dos dados, mas com a vantagem de permitir acessos de leitura e de escrita a áreas de outras partições destinadas a outras IMs. As áreas possíveis de aceder são adjacentes à partição obtida por um dado *thread*. A sintaxe para aplicar uma política de distribuição **distshared** é:

$$\textbf{distshared}(e_1, e_2)\ \textit{variable\_declaration}$$

onde e_1 e e_2 são inteiros que indicam o número de posições das partições adjacentes a que se quer obter acesso, sendo esta notação apenas aplicável a vectores.

Tal como se referiu anteriormente, o construtor **distshared** oferece a capacidade de aceder a posições que não estão na partição de dados dada ao *thread*, o exemplo ilustrado na listagem 3.7 contém a criação dum método, no qual queremos substituir as ocorrências

de uma dada *substring* num vector de caracteres. O construtor **distshared** precisa que lhe sejam indicados dois parâmetros, o número de posições precedentes e posteriores à partição a que ser quer ter acesso. No exemplo indicado queremos ter acesso a zero posições antes e a até quantas posições necessárias depois, de modo a que uma ocorrência da *substring* comece dentro dos limites definidos pela partição dos dados.

Listagem 3.7: Substuição de todas as referências duma *substring* num texto

```
1  public char[] replace(String oldStr, String newStr, distshared(0,
       oldStr.length-1) char[] text) reducewith ArrayAssembler<Char>() {
2  for (int i = 0; i < text.length-(oldStr.length-1);  i++) {
3    String s = "";
4    for (int j = 0; j < oldStr.length; j++)
5      s = s + text[j+i];
6    if (s.equals(oldStr))
7    for (int j = 0; j < oldStr.length; j++)
8      text[j+i] = newStr.charAt(j);
9  }
10 return text;
11 }
```

A figura 3.4 exemplifica o caso particular da substituição da *string* "aaa" pela *string* "xxx". Neste caso o número de posições de outras partições a que cada instância do método tem acesso são duas, encontrando-se estas no final da sua partição.

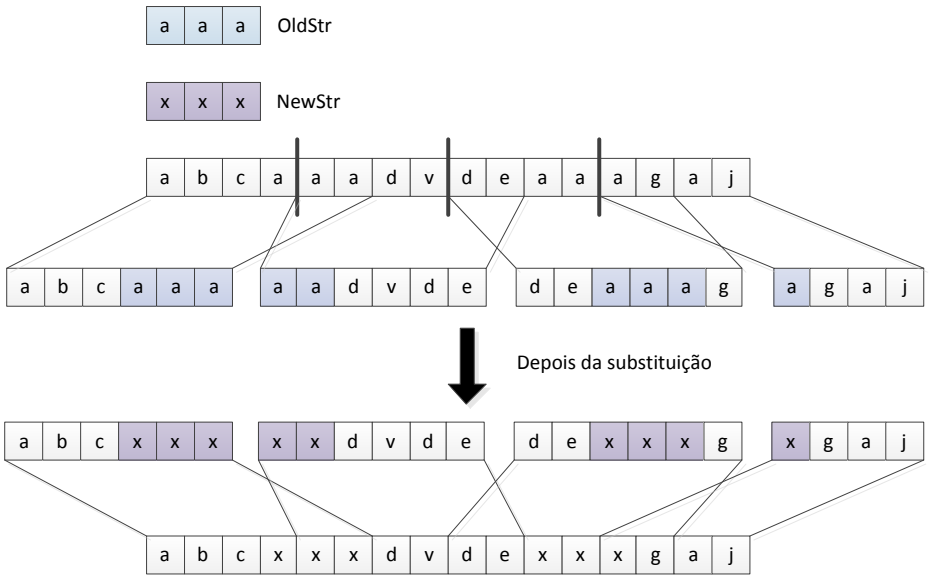


Figura 3.4: Exemplo da aplicação do construtor *distshared*

Apresentados o modelo de execução SOMD, o paradigma de programação DMR e os construtores que dão suporte ao modelo de execução, será descrita no próximo capítulo a implementação realizada numa extensão à linguagem Java.

4

Arquitectura e Implementação

Este capítulo é dedicado à abordagem utilizada para incorporar o modelo de execução SOMD na forma do paradigma DMR na linguagem de programação Java. Será descrita a arquitectura da solução encontrada, a sintaxe concreta da extensão, a integração realizada com o *runtime* do X10 e por fim, serão descritas as transformações realizadas ao código e a sua geração.

4.1 Arquitectura

Como referido no capítulo 3, o modelo foi instanciado como uma extensão à linguagem Java. Para tal foi utilizada a ferramenta Polyglot [NCM03], que permite criar compiladores para extensões à linguagem Java. A implementação das funcionalidades exigidas aos construtores foi realizada recorrendo-se ao sistema de execução da linguagem X10. Este apresenta uma biblioteca Java, podendo assim ser integrado em computações desta linguagem.

Relembrando o modelo de execução do X10, este está organizado em *localidades* (*places*), sendo estes uma porção do espaço de memória particionado. Acrescido ao conceito de *place* existe o conceito de *actividade*, computações assíncronas que são executadas nos *places* onde são lançadas. O modelo APGAS permite realizar o seu mapeamento para o modelo de execução SOMD, podendo execução ser realizada em um ou mais *places*, consoante se pretenda um ambiente de memória partilhada ou memória partilhada distribuída. Por omissão a execução ocorre num único *place*.

A figura 4.1 apresenta o processo de transformação do código presente nos ficheiros com a extensão *sp* em ficheiros com a extensão *java*, com referências à biblioteca Java do X10. Os ficheiros de entrada são processados pelo compilador gerado pelo Polyglot,

que por sua vez realizará transformações ao código nos seus diversos passos, de modo a produzir neste processo código Java que ao ser compilado para *bytecode* é reconhecido pela Java Virtual Machine (JVM).

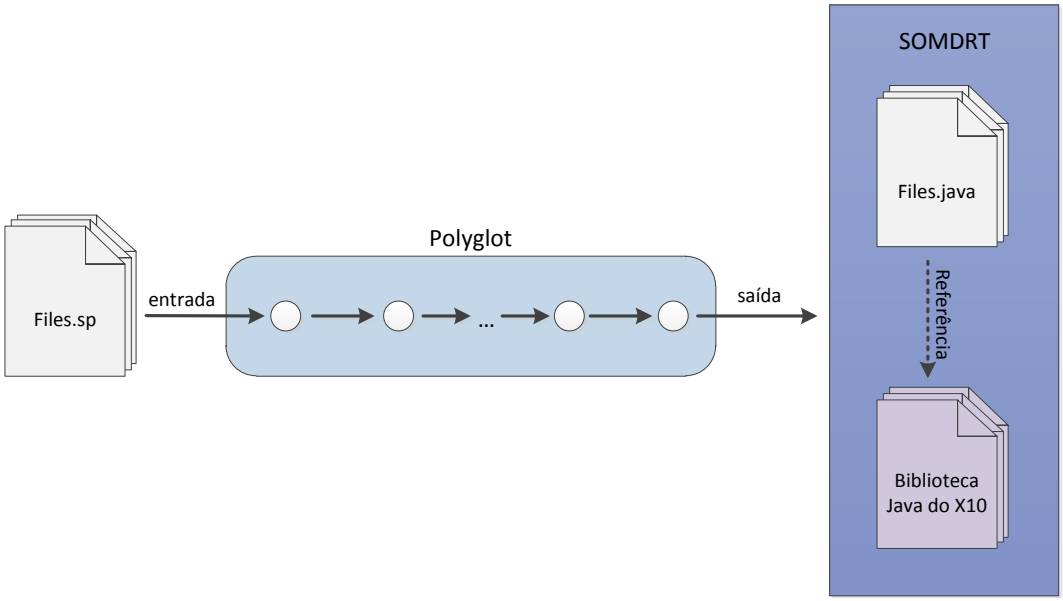


Figura 4.1: Processo de compilação

De seguida descrevemos de forma sucinta a ferramenta Polyglot, indicando também o seu modo de funcionamento.

4.1.1 Polyglot

O Polyglot é uma ferramenta que permite criar compiladores *source-to-source*, que recebem como entrada código numa linguagem e geram código noutra. Neste caso em particular, recebe como entrada ficheiros duma dada extensão de linguagem e que produz como resultado ficheiros com código Java que são compilados usando um compilador Java.

O compilador submete os ficheiros de entrada a uma sequência configurável de passos, até produzir o código Java final. Um passo é uma dada actividade do processo de compilação, que segue o modelo de *visitor* [PJ98], os nós da *Abstract Syntax Tree* (AST) são processados em cada passo, sendo a AST resultante de cada passo, submetida ao passo seguinte. O conjunto base de passos do compilador são:

parsing - cria uma *Abstract Syntax Tree* (AST) a partir da gramática reconhecida pelo analisador sintáctico criado com o *Polyglot Parser Generator* (PPG), uma extensão ao CUP [HFA99], um gerador de analisadores sintácticos LALR para Java. O PPG permite extender uma gramática de linguagem existente no CUP ou no PPG, com modificações localizadas, evitando ter que reescrever uma gramática por completo;

build-types - constrói um objecto Type que representa cada tipo existente no ficheiro de entrada;

clean-super e clean-sigs - remoção de ambiguidades encontradas na declaração de supertipos e na declaração de métodos de classes e interfaces;

add-members - recolhe informação sobre os membros duma classe ou interface e adiciona-a aos objectos Type criados no passo *build-types*;

disambiguate - remove ambiguidades existentes nos corpos dos métodos, construtores ou inicializações;

type checking - realiza uma análise semântica ao código;

exception checking - realiza uma análise semântica à declaração de excepções e à sua propagação;

reachability checking - verifica se não existem porções de código que não sejam alcançáveis dentro dum método;

exit checking - verifica se todos os caminhos existentes num método que deve retornar um dado valor, retornam esse valor;

initialization checking - verifica se as variáveis locais estão inicializadas antes de serem utilizadas;

translation - percorre os nós da AST para escrever o código final no ficheiro alvo.

Quando se invoca o compilador do Polyglot indica-se qual a extensão que se quer processar, de modo a que este carregue a classe fonte ExtensionInfo correspondente. Esta classe contém informações relativas à extensão dos ficheiros de entrada, ao gerador de nós da AST, sistema de tipos e escalonamento dos passos. Aos passos existentes numa extensão podem ser adicionados novos passos e modificada a ordem de escalonamento destes.

4.2 Sintaxe concreta

A extensão realizada é uma extensão à versão 5 da linguagem Java[1] em que é permitido o uso dos construtores definidos na secção 3.3. A implementação da extensão requereu alterações de produções da gramática e a implementação dos nós respectivos nós de AST. As modificações introduzidas focaram-se a dois níveis:

[1]O compilador Polyglot para a versão 5 do Java foi construído como uma extensão à versão base do Polyglot

Tabela 4.1: Sintaxe do cabeçalho do método

<cabecalho_metodo>	::=	<modificador> <tipo_retorno> <nome_metodo> **(** [<lista_parametros>] **)** [<reducewith>]
<reducewith>	::=	**reducewith** <construtor>
<lista_parametros>	::=	<parametro> [**,** <lista_parametros>]
<parametro>	::=	[**dist**] <tipo> <nome>
	|	**dist :** <construtor> <tipo> <nome>

Alterações sobre os cabeçalhos dos métodos A tabela 4.1 resume as modificações efectuadas à linguagem na representação Extended Backus-Naur Form (EBNF), para acomodar a inclusão dos novos construtores ao nível do cabeçalho dos métodos. A utilização de parêntesis rectos nesta notação indica a opcionalidade da construção. A este nível foi permitido o uso do construtor **reducewith** no final da declaração do método, sendo este sucedido do construtor da classe que implementa a política de redução. A adição deste construtor é opcional, uma vez que a métodos do tipo void não se aplica qualquer redução, sendo esta situação verificada semanticamente.

Foi também necessário permitir a qualificação dos parâmetros do método com o modificador **dist**. O construtor é seguido do construtor da classe que implementa a política de distribuição, e à semelhança **reducewith** a indicação da política pode ser omitida, em detrimento duma política por omissão sobre vectores.

Alterações sobre o corpo dos métodos A listagem 4.2 resume as modificações realizadas para permitir o uso dos novos construtores, ao nível do corpo dos métodos. Estes são métodos Java normais cuja implementação pode fazer uso dos seguintes construtores: **dist**, **shared**, **barrier** e **atomic**.

O código presente no corpo dum método método SOMD será executado em paralelo, contudo existem construtores que não se adequam ao contexto duma execução paralela. Exemplos paradigmáticos deste caso são os construtores **dist** e **shared**. No caso do **dist** a declaração duma variável deste tipo, no contexto de uma execução paralela do programa, equivale à declaração duma variável local, pois os restantes IMs não teriam acesso às partições a si destinadas, uma vez que todos criaram essa variável no seu escopo. O caso do **shared** é análogo ao do **dist**. Aquando da execução paralela do programa a criação deste tipo de variáveis, faria com que todos os IMs criassem uma variável deste género no seu escopo, o que na realidade corresponderia à criação duma variável local que seria apenas usada pelo *thread* que a criou.

Assim sendo a declaração de variáveis **dist** e **shared** está restringida ao início do programa, numa zona na qual as operações são efectuadas de forma sequencial. O corpo do método apresenta assim duas secções: a primeira relativa ao início do corpo onde apenas podem ser declaradas variáveis do tipo **dist** e **shared**, e a segunda que corresponde ao código que será executado em paralelo.

A definição de corpo do método foi então estendida de modo a permitir o uso dos

Tabela 4.2: Sintaxe do cabeçalho do corpo do método

extends	<corpo_metodo>	::=	{ [<decl_dist_shared_vars> [<lista_construcoes>] }
	<decl_dist_shared_vars>	::=	<decl_dist_shared> [<decl_dist_shared_vars>]
	<decl_dist_shared>	::=	**dist** <decl_init_variavel> **;**
		|	**shared** <decl_init_variavel> **;**
	<lista_construcoes>	::=	<construcao> [<lista_construcoes>]
extends	<construcao>	::=	**barrier ;**
		|	**atomic** { [<lista_construcoes>] }

construtores SOMD, sendo distinguidas duas secções: uma referente apenas à declaração de variáveis **dist** e **shared**; e outra secção respeitante à execução paralela e que contém construções Java. Os construtores **atomic** e o **barrier** foram introduzidos como uma extensão às construções permitidas pela linguagem. Em relação ao **atomic** este é aplicado a um bloco de código, ou seja, a uma sequência de construções.

Os nós AST necessários para implementar as alterações na gramática definidas acima, encontram-se descritos na tabela 4.3.

Tabela 4.3: Descrição dos nós de AST introduzidos na gramática

AtomicBlock	Representa o construtor **atomic**, guardando o bloco de operações a realizar de forma atómica
Barrier	Denota o construtor **barrier**
DistParameter	Representa um parâmetro que faz uso do construtor **dist**
DistVar	Denota a declaração duma variável local distribuída através do uso do construtor **dist**
SOMDBlock	Identifica o código que irá ser executado segundo o modelo de execução SOMD. O conteúdo deste apresenta duas partes, a primeira relativa à declaração de variáveis locais SOMD, e a segunda referente ao código que executará em paralelo e que poderá apresentar os construtores **barrier** e **atomic**
SharedVar	Análogo ao nó DistVar, mas tratando-se neste caso de variáveis locais partilhadas através do uso do construtor **shared**
SpringMethodDecl	Usado para representar a declaração dos cabeçalhos dos métodos SOMD. Guarda informação sobre qual a redução a aplicar, os parâmetros do método e o corpo do método

4.3 Compilação para X10

Finda a introdução dos construtores na gramática, apresenta-se de seguida o mapeamento do modelo de execução SOMD nos construtores da linguagem X10, isto tendo em vista a transformação dos nós SOMD em nós Java com chamadas à biblioteca Java do X10.

Para aferir-se que construções X10 seriam mais adequadas para a implementação das funcionalidades exigidas aos construtores, foram realizados pequenos programas na linguagem X10, cujo comportamento simula as funcionalidades a implementar. O foco é nos ambiente *multi-core* mas tendo-se em conta também, uma possível extensão do suporte do modelo para ambientes distribuídos. A análise efectuada originou as conclusões apresentadas ao longo desta secção.

Modelo base - Ambientes de memória partilhada

No que se refere ao modelo DMR base, o código X10 gerado apresenta a mesma decomposição em três fases. A figura 4.2 apresenta de forma resumida o que foi mencionado na secção 3.2 para um sistema *multi-core*. A fase de distribuição dá origem a um vector de elementos contendo as várias partições da estrutura de dados. Esta estratégia têm em conta o número de *threads* disponíveis, na altura em que se particiona os dados. A distribuição dos dados implica a cópia das zonas de memória dos dados para criar as partições, seja qual for a distribuição a utilizar.

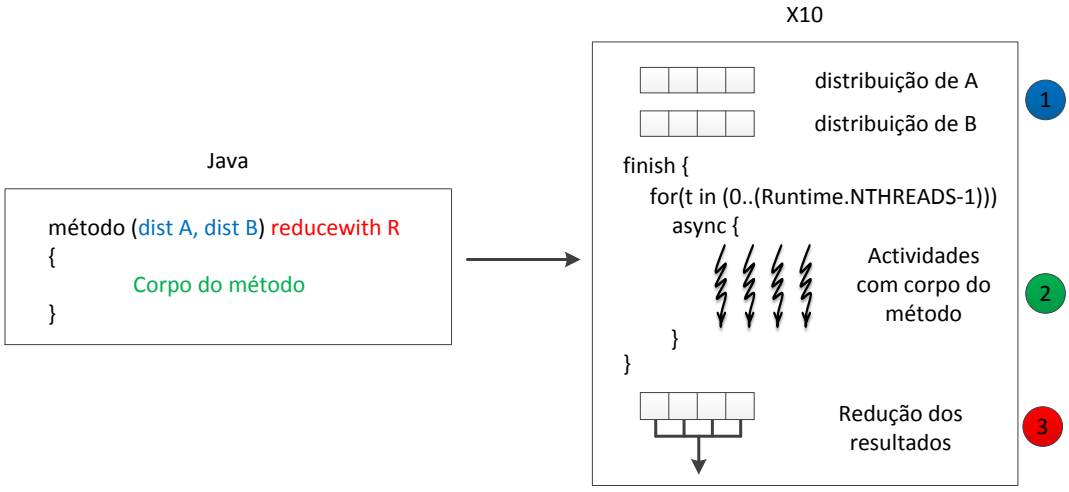


Figura 4.2: Modelo SOMD num programa X10 num sistema *multi-core*

Nesta fase realizaram-se experiências com vectores nativos Java e vectores X10, uma vez que a linguagem X10 apresenta uma implementação própria desta estruturas, de modo a aferir a possibilidade da utilização dos primeiros no sistema de execução X10, e de qual a estrutura de vectores que menor impacto provoca no desempenho dos programas. Constatou-se que é possível utilizar-se vectores Java em computações X10, tendo-se realizado a partição de dados e a respectiva distribuição dos mesmos pelos *threads*. Com isto queria-se observar se era obrigatório usar os vectores X10, o que implicaria a fases de transformação de vectores Java para os seus congéneres do X10, e que se traduziria em custos acrescidos a nível de desempenho.

O suporte à execução paralela das várias instâncias de um dado método foi implementado sobre as *actividades* do X10, uma *actividade* por instância. A sua execução é delimitada por um bloco **finish** que impõe um ponto de sincronização para que a execução do restante código (estágio de redução) só prossiga quando todas as *actividades* tiverem terminado.

Cada *actividade* executará o código referente a uma instância do método, trabalhando cada uma sobre as suas partições de dados, produzidas na fase de distribuição. O número de *actividades* a lançar depende do tamanho, determinado em tempo de arranque, da *pool* de *threads* do X10. Por forma a que a execução das IMs não seja sequencializada, o tamanho da *thread pool* não deverá ser superior ao número de processadores disponíveis para a computação.

O número de *actividades* deverá tomar um valor máximo igual ao número de processadores disponíveis, de modo a prevenir perdas de desempenho devido ao escalonamento de tarefas.

Os resultados parciais computados por cada instância do método são colocados num vector, de forma a serem disponibilizados ao estágio de redução. Para a construção deste vector recorreu-se a um vector partilhado por todas as *actividades*, sendo que cada uma escreve numa posição pré-determinada desse vector.

O estágio de redução é executado assim que termina o bloco **finish**. A sua implementação pode ser realizada de duas formas distintas:

- Uso da função reduce existente nas estruturas de dados de vectores do X10, nomeadamente Arrays. A operação reduce recebe como parâmetro uma função que deve ser aplicada à estrutura de dados;
- Aplicação de um método especificamente implementado para realizar a redução desejada.

A solução adoptada foi a segunda, pois permite um maior desempenho que não se consegue com a solução usando os vectores X10, uma vez que o uso destes implica estágios de transformação de vectores Java em vectores X10 o que penaliza o tempo de execução. Além de que a função a usar pelo método reduce teria de ser mapeada numa classe de fecho do sistema de execução X10 (para mais detalhes consultar secção 4.4), enquanto a segunda solução apenas implica a criação de classes que implementem a interface definida para a redução.

Modelo base - Ambientes de memória distribuída

No que se refere a ambientes de memória distribuída, foram novas estratégias para a distribuição e para a redução. Assim foram testados os DistArrays para particionar os dados ao nível dos *places* e uso de GlobalRefs para armazenar os resultados parciais das *actividades* nos diversos *places*, de forma a possibilitar a realização da operação de redução.

O uso dos DistArrays permite a aplicação de funções de distribuição a nível de *places* de forma automática, através dos métodos makeBlock, makeConstant, make(region), entre outros presentes na classe Dist. Esta classe é a classe responsável por definir a distribuição a usar pelo DistArray. A nível do particionamento dos dados pelas *actividades* de cada *place*, as estratégia utilizada quando se usou DistArrays passou pela estratégia adoptada para ambientes de memória partilhada. Apesar desta alternativa para realizar o particionamento dos dados, optou-se por manter uma solução baseada nos vectores Java, uma vez que o uso dos DistArrays implica à semelhança dos vectores X10, estágio de transformação de dados que penalizam o desempenho do programa.

Na fase de execução paralela é necessário enviar a computação para os diferentes *places* que compõem o ambiente distribuído. Para tal foi utilizado o construtor **at** para enviar a computação para os *places* disponíveis. Em cada *place* à semelhança dum sistema *multi-core* são lançadas um número de *actividades* igual ao número de *threads* existentes no sistema. A listagem 4.1 apresenta o que foi referido anteriormente.

Listagem 4.1: Execução em ambientes distribuídos na linguagem X10

```
1  finish {
2    for(place in Place.places()) at(place) {
3      for(t in (0..(Runtime.NTHREADS-1))) async {
4        \\codigo a executar
5      }
6    }
7  }
```

A implementação da redução em sistemas distribuídos é semelhante à usada em sistemas *multi-core*, com a diferença que é usada uma variável partilhada, um GlobalRef, para guardar o vector de resultados. Este vector apresenta o tamanho de $threads \times places$ no *place* inicial, o que permite que cada *actividade* de cada *place* escreva na sua posição correspondente do vector, o valor por si computado. Depois de finalizado o bloco **finish** é aplicada a estratégia de criação dum método para aplicar a redução aos resultados existentes no vector do GlobalRef. Este foi preferido em detrimento do DistArray, pois este último obrigaria a no final a aceder a cada um dos *places* onde os resultados do DistArray estão armazenados de modo a que possam ser passados à fase de redução. O GlobalRef por seu lado simplifica este processo pois os dados estão todos guardados no mesmo *place*, sendo apenas necessário enviar o vector de resultados parciais à função de redução. Em todas as abordagens de redução apresentadas, os vectores que guardam os resultados são sempre do tipo de retorno do método SOMD.

Variáveis locais partilhadas

Tal como foi referenciado na secção 4.2 é permitida a declaração de variáveis locais **shared** e **dist**, no início do método. O mapeamento duma variável local **dist** em código X10 é feito tal como os parâmetros **dist**, ocorrendo uma fase de distribuição dos dados antes da

execução do método, contudo esta só é realizada após as distribuições dos parâmetros.

Tal como indicado no capítulo anterior, os parâmetros do método (à excepção dos **dists**) são considerados partilhados. Porém isto só é válido para arquitecturas *multi-core*, em ambientes distribuídos tal é apenas conseguido através do construtor **shared**. Em relação ao construtor **shared** este foi mapeado num GlobalRef que guarda o conteúdo da variável no *place* em que está localizada o *thread* Java que realiza a computação. Este é declarado na mesma zona do código X10 dedicada às distribuições, ou seja antes da execução paralela. Um GlobalRef é visível por todas as *actividades* permitindo o acesso a uma variável que deste modo se encontra partilhada por estes. A implementação do **shared** mantém-se igual seja qual for o ambiente de execução que se esteja a usar.

A figura 4.3 mostra a declaração de uma variável local distribuída e de uma variável local partilhada, sendo esta última acedida para leituras e escritas por todas as *actividades*.

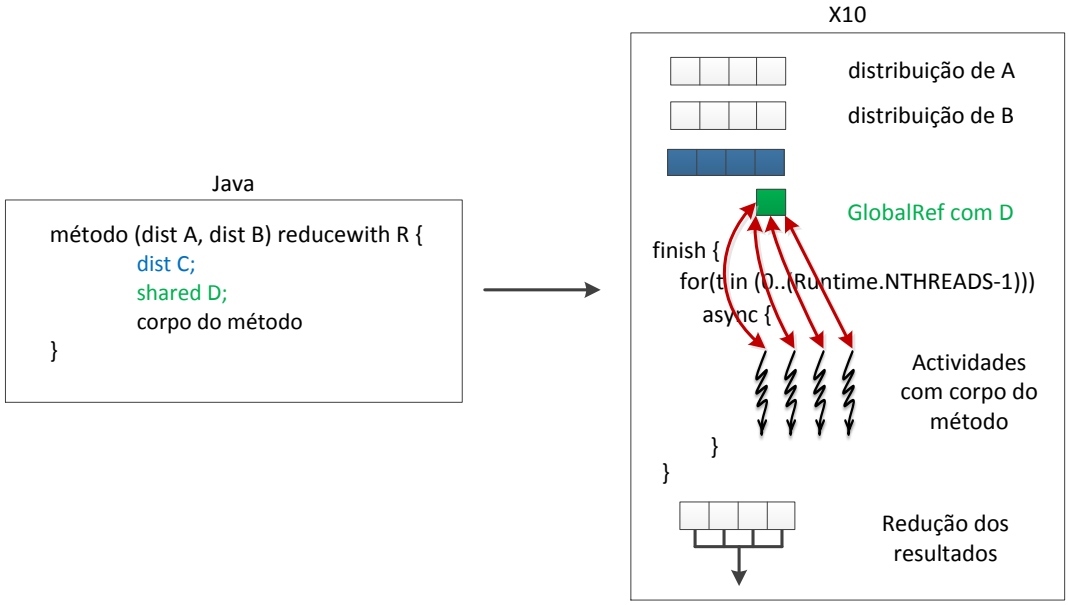


Figura 4.3: Declaração de variáveis locais dum método SOMD num programa X10 (retirado de [MP12])

Sincronização

O construtor **atomic** é mapeado trivialmente no seu congénere da linguagem X10, *atomic*.

Já para simular o uso do construtor **barrier** foram usados os *clocks* do X10. Cria-se um Clock antes do bloco **finish** respeitante à execução do código em paralelo e no lançamento das *actividades* qualificam-se estas com o construtor **clocked***(clock)*, para indicar que as *actividades* lançadas devem ficar registadas no Clock criado. Isto permite assim o controlo sob a execução das *actividades* de modo a sincronizá-las. A sincronização nos pontos do código indicados com **barrier** passam a ser chamadas ao método clock.advance() que faz

com que as *actividades* bloqueiem nesse ponto até que todas as *actividades* registadas no *clock* cheguem a esse ponto.

A figura 4.4 mostra o mapeamento que é realizado para implementar o construtor **barrier** na linguagem X10. As linhas que separam a execução das *actividades* são as chamadas ao método clock.advance(), que servem para sincronizar as várias *actividades* em execução.

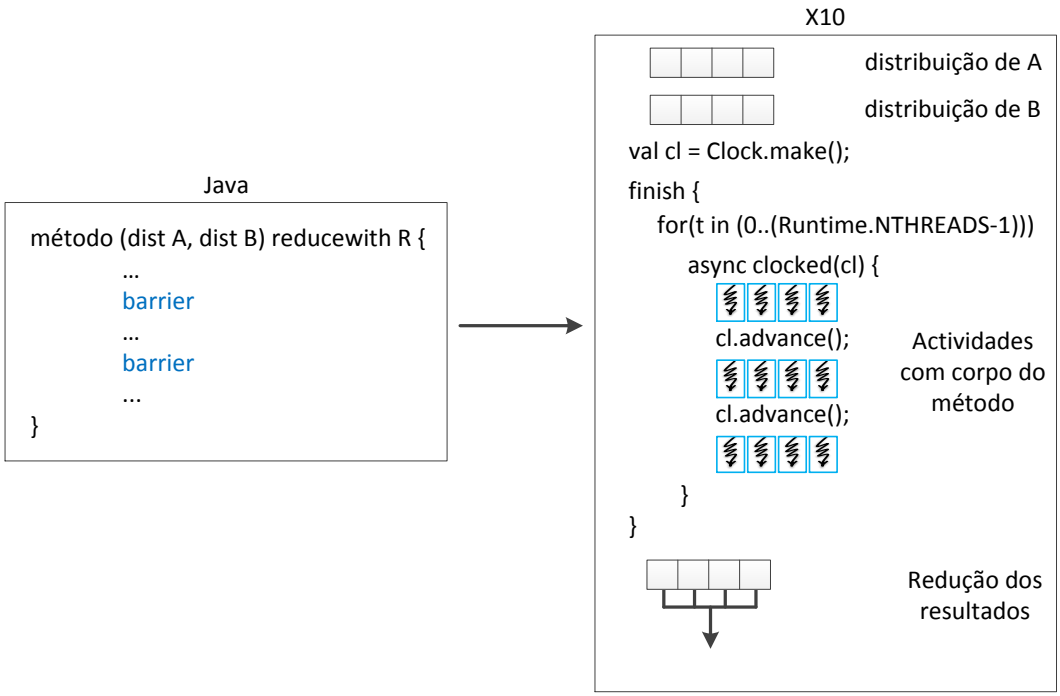


Figura 4.4: Uso de barreiras dum método SOMD num programa X10

Distshared

Este construtor ao contrário dos anteriores não dispõe de uma classe ou construtor que consiga simular o seu comportamento directamente na linguagem X10. Assim a maneira encontrada para simular o seu comportamento foi através do uso de DistArrays, em que cada acesso é precedido da verificação se a posição aceder pertence ou não à partição adjudicada à *actividade*. Caso não pertença a execução é enviada para o *place* que alberga a posição a aceder - *actividades* X10 só podem aceder a zonas de memória do seu *place*.

Em trabalho exploratório foi implementada uma classe DistShared baseada no código Java da classe DistArray da biblioteca do X10, de modo a encapsular o comportamento desejado numa estrutura de dados, tendo-se verificado que funcionava pelo menos com vectores.

4.4 Arquitectura da solução para a integração das linguagens

Elaborado este trabalho de análise sobre as funcionalidades permitidas pelo X10, analisou-se o código gerado pelo compilador da linguagem X10 para os exemplos que simulam o comportamento dos construtores a implementar. Esta análise teve por objectivo encontrar a melhor forma de mapear os construtores SOMD na API do sistema de execução do X10 em Java (X10JRT).

A compilação de um ficheiro X10 gera um ficheiro Java que contém uma classe com informações relativas: ao modelo de serialização usado pelo X10; os métodos usados na classe X10; uma classe estática que é utilizada para executar o código X10, que funciona como classe Main; várias outras classes estáticas que definem *closures* (fechos) das computações encapsuladas por cada *at* e *async* presentes no código X10.

Dada esta estrutura, decidiu-se que cada método SOMD seria mapeado numa classe que, à semelhança do X10, encapsularia os fechos das acções da execução paralela (*closures* de *at* e *async*).

Tendo em vista a desacopulação da especificação da implementação foram definidas interfaces para as classes que representam métodos SOMD, distribuições e reduções.

A interface SOMD, apresentada na listagem 4.2, especifica os métodos que devem ser implementados pela classe que encapsula a informação relativa ao método SOMD:

- init - construtor da classe que recebe como parâmetros as variáveis que devem ser passadas ao método SOMD;
- run - inicia a execução do método SOMD com todas as suas fases (distribuição, execução e redução);
- getResult - devolve o resultado da execução do método SOMD.

O método run não retorna o resultado da computação porque a sua execução é delegada num trabalhador, *thread* da *pool* do X10JRT. Uma vez terminada a computação, o resultado pode ser obtido através do método getResult.

Listagem 4.2: Interface SOMD

```
public interface SOMD<T> extends x10.x10rt.X10JavaSerializable {
  public void run();

  public SOMD<T> init(Object... objects);

  public T getResult();
}
```

A especificação das políticas de distribuição implementadas pelo utilizador é definida na interface Distribution. A política deve ser definida dando uma implementação concreta do seu único método, dist, que recebe como parâmetro a estrutura de dados que se quer

particionar e o número de partições a realizar[2], retornando um vector com as partições realizadas. A listagem 4.3 apresenta a interface Distribution.

Listagem 4.3: Interface Distribution

```
public interface Distribution<T> {

  T[] dist(T x, int partitions);

}
```

Para especificar as reduções foi criada a interface Reduction, apresentada na listagem 4.4. Esta classe especifica apenas o método reduce que define a política de redução a utilizar. Este método recebe o vector de resultados proveniente da execução do método e aplica a função de redução a esse vector de modo a obter o resultado final.

Listagem 4.4: Interface Reduction

```
public interface Reduction<T> {

  T reduce(T[] array);

}
```

As funcionalidades comuns a qualquer código gerado foram factorizadas num conjunto de classes abstractas que definem o sistema de execução SOMD sobre o X10. Dessas classes constam as classes Closure e SOMDEnqueuer. Estas no seu geral factorizam informação sobre super-classes e interfaces, relativas ao sistema de execução X10 assim como métodos e campos referentes à serialização de dados do X10. Destas salienta-se a classe SOMDEnqueuer, apresentada na listagem 4.5. Nas linhas 5 a 7 encontra-se campos importantes, que intervêm na integração dos sistemas de execução X10 e Java. Nesta listagem é ainda apresentada a implementação do método enqueue que coloca a própria instância da classe na fila de execução de métodos SOMD, abordada mais adiante nesta subsecção.

A execução dum método SOMD no sistema de execução X10 só pode ser realizada à custa de uma *thread* X10. Contudo queremos fazer a integração deste modelo de execução com computações que ocorrem em *threads* Java, tendo de para isso de ligar os dois sistemas de execução.

Ao analisar o código gerado pelo compilador do X10, verificou-se a existência duma classe que serve de ponto de entrada para a execução dos programa X10. Esta é uma classe interna (*inner class*) que estende x10.runtime.impl.java.Runtime e tem por objectivo iniciar o sistema de execução do X10 transformando a *thread* Java em execução numa *thread* X10 e, de seguida, executar o código do método runtimeCallback. Este último, por

[2]Apesar deste ser pré-determinado em tempo de arranque, a construção da classe de distribuição necessita de receber esta informação de forma a evitar chamadas ao sistema de execução X10 para obter essa informação.

Listagem 4.5: Classe abstracta SOMDEnqueuer

```
1  public abstract class SOMDEnqueuer<T> extends x10.core.Ref
2      implements SOMD<T> {
3    private static final long serialVersionUID = 1L;
4
5    protected Lock lock;
6    protected Condition resultQueue;
7    protected boolean hasResult;
8
9    public x10.rtt.RuntimeType<?> $getRTT() {...}
10
11   public void $_serialize(x10.x10rt.X10JavaSerializer $serializer)
12       throws java.io.IOException { }
13
14   protected x10.array.Dist getDistribution(final int size) {...}
15
16   public SOMDEnqueuer(final java.lang.System[] $dummy) {
17     super($dummy);
18   }
19
20   public void enqueue() {
21     XRT.SOMDRuntime.enqueue(this);
22   }
23
24 }
```

sua vez, invoca o método main do programa e uma vez finalizado, termina a execução na máquina virtual (System.exit(0)).

Pretende-se que o sistema de execução do X10 seja capaz de processar os pedidos de execução de métodos SOMD de várias *threads* e portanto não termine a execução da máquina virtual após o método *runtimeCallback* ser terminado.

Para tratar o problema relativo ao processamento dos múltiplos pedidos, procedeu-se à alteração da classe x10.runtime.impl.java.Runtime da biblioteca do X10, dividindo o método que inicia o sistema de execução deste e dá início à execução do código do método *main* em três partes: inicialização, execução e terminação do sistema de execução. Assim só com a chamada do método para terminar o sistema de execução é que a máquina virtual é finalizada.

Em relação ao processamento de pedidos por parte de várias *threads*, optámos pela criação duma fila de trabalhos, em que as *threads* Java assumem o papel de produtoras, colocando pedidos de execução, e a *thread* X10 assume o papel de consumidora, retirando e processando os pedidos. A figura 4.5 retrata o funcionamento da integração feita entre os sistemas de execução Java e X10 através da fila de trabalhos. Na figura está ilustrada apenas uma *thread* Java a colocar pedidos na fila, porém podem ser mais.

Da compilação destas soluções resultou a classe SOMDRuntime que faz a ponte entre os sistemas de execução Java e X10. Esta classe à semelhança da *inner* classe que serve

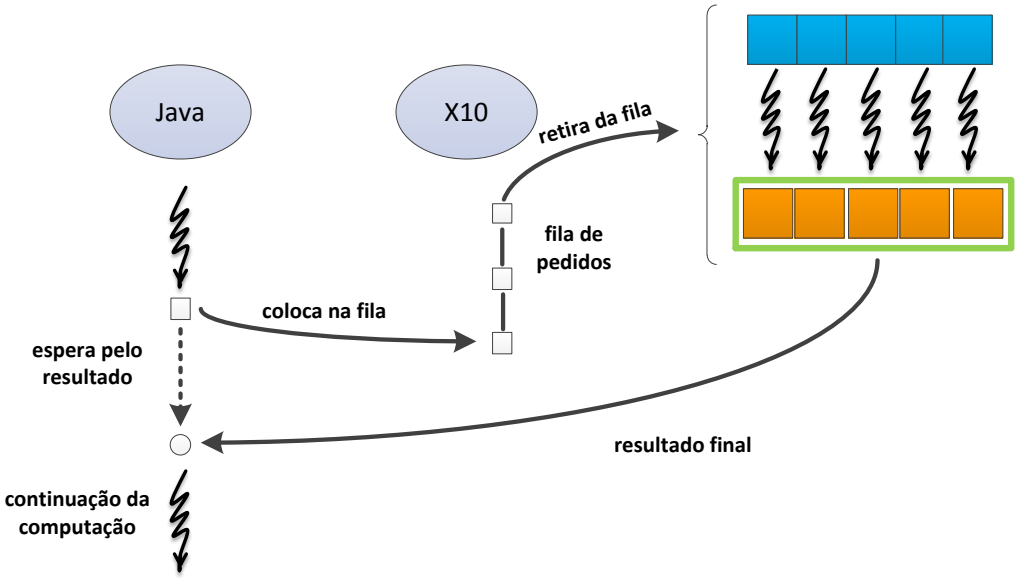


Figura 4.5: Integração dos sistemas de execução Java e X10 (retirado de [MP12])

classe Main, extende a classe x10.runtime.impl.java.Runtime e apresenta uma fila de trabalhos a executar, com instâncias das classes SOMD.

A figura 4.6 apresenta as fases do programa e chamada dos métodos criados na classe SOMDRuntime. O início do programa a executar é seguido pela criação de uma *thread* Java, para executar o sistema de execução do X10, ficando a *thread* original encarregue do código da aplicação.

Uma invocação de um método SOMD assegura-se que o sistema de execução do X10 está pronto a receber pedidos de trabalho através do método waitInit. Finalizado este passo, a *thread* Java invocadora pode colocar o seu pedido de execução na fila de trabalhos e ficar a aguardar pelo resultado da computação do método, que apenas será obtido quando o pedido for processado por parte da classe SOMDRuntime.

A *thread* responsável pela execução do sistema de execução do X10 inicia o seu trabalho ao carregar o sistema de execução, de seguida passa para um estágio de sincronização de *threads* referente à utilização de múltiplos *places* devido ao lançamento de várias JVM.

Terminada a fase de inicialização a *thread* Java que despoletou a execução do X10JRT transforma-se e dá origem a um *thread* X10, pronto a receber e executar pedidos de execução de métodos SOMD. Neste instante, o *thread* original Java bloqueado no método waitInit é notificada (através de um invocação do método isReady) e prossegue o seu trabalho. A *thread* X10 prossegue por sua vez para a execução do método runtimeCallBack continuamente retira da fila de trabalho instâncias de classes SOMD, sendo lançada a sua execução, sendo apenas retirada uma instância de cada vez, pois cada uma faz uso da totalidade de trabalhadores existentes na *pool* do X10. A execução do *thread* X10 é na figura

ilustrada pelos pontos escuros do diagrama.

No final da execução do programa Java é indicado ao *runtime* do X10 que a sua execução pode terminar, através da invocação do método terminate. A invocação deste método provoca a terminação do método runtimeCallback, permitindo ao fluxo inicial prosseguir e terminar a execução do programa (através do método exitRT).

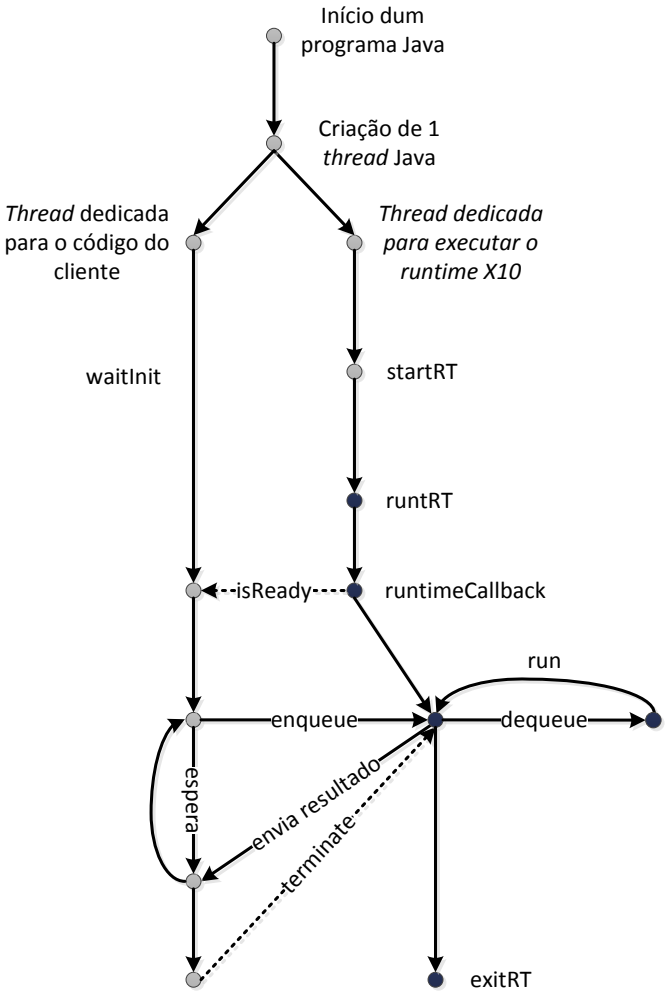


Figura 4.6: Diagrama da integração entre as *threads* Java e X10

Definida a estrutura das classes que vão ser usadas neste modelo, procedemos à geração de código e às transformações que devem ser realizadas, para se executar um método segundo este modelo.

4.5 Implementação do Compilador

A implementação do compilador requereu a introdução de três passos no processo de compilação da *framework* Polyglot: o *SpringRewrite*, *SpringX10Replacer* e o *SpringX10-SOMDPrettyPrinter*.

4.5.1 Passo SpringRewrite

Este passo é executado após o passo *buildTypes* e tem como funções: transformar métodos classificados como privados em protected; explicitar o uso do this no uso de membros de dados da classe que são invocados ou acedidos; e uniformizar nós AST relativos ao corpo de métodos SOMD (que podem ser Block ou SOMDBlock) em nós SOMDBlock.

As transformações de AST referentes às duas primeiras indicadas acima, prendem-se com o facto de se fazer uso de membros e métodos da classe que que os alberga. Para que tal propriedade se mantenha no código gerado é necessário passar uma referência do objecto que contém esse mesmo método, uma vez que método SOMD vai ser movido para uma classe de fecho que não tem acesso a esses campos ou métodos.

Esta abordagem não é a ideal, pois em ambientes de memória distribuída um objecto pode conter demasiada informação que deverá ser serializada, contudo em ambientes *multi-core* onde objectos são passados por referência esta não apresenta um impacto significativo na execução do programa. A solução ideal a usar em ambientes distribuídos deveria passar pela análise de quais os membros de dados e que métodos são invocados. No caso dos membros de dados apenas aqueles que são efectivamente usados deveriam ser copiados pela rede, e no caso dos métodos estes deveriam ser copiados para a classe Closure referente à execução da *actividade*.

A transformação de nós Block para SOMDBlock é importante para focar o trabalho dos passos seguintes, uma vez que nós classificados como SOMDBlock correspondem a código que deverá ser executado em paralelo.

4.5.2 Passo SpringX10Replacer

Este passo é responsável por realizar transformações nos nós SOMDBlock da AST e recolher informações necessárias para a geração de código implementada no passo SpringX10SOMDPrettyPrinter (subsecção 4.5.3).

No caso das transformações, estas adicionam chamadas a métodos e membros de dados da X10JRT e também referências a variáveis que só serão criadas no passo que gera as classes SOMD. Isto é realizado tendo em vista a cópia do código modificado para as classes que serão geradas no passo seguinte. As transformações aos nós AST são efectuadas apenas em nós pertencentes ao corpo de métodos SOMD. Tal é verificado testando se os nó referentes ao corpo de métodos nos nós SpringMethodDecl [3] se tratam de nós SOMDBlock.

As informações relevantes a cada método SOMD que são extraídas neste passo dizem respeito: ao uso da redução a aplicar em cada método; que variáveis são partilhadas; e que variáveis são distribuídas e as suas distribuições. Toda a informação recolhida será utilizada na geração de código efectuada no passo SpringX10SOMDPrettyPrinter.

De seguida são descritas as transformações realizadas e as recolhas de informação efectuadas nos nós de AST relevantes visitados neste passo:

[3] Declarações de métodos do modelo dos serviços que estende o Java.

SpringMethodDecl Caso corresponda a um cabeçalho dum método SOMD são recolhidas neste nó as informações sobre a redução, nome do método, parâmetros (no caso de serem distribuídos obtém-se também informação sobre as suas distribuições) e código do corpo do método resultante da transformações efectuadas;

DistVar Recolha da informação relativa à distribuição a aplicar para esta variável;

LocalAssign Um nó LocalAssign denota a operação de uma atribuição realizada a uma variável. A verificação destes nós só é relevante quando se está a verificar atribuições a variáveis partilhadas. Nesse caso é recolhida informação que levará à geração duma nova classe, para realizar a atribuição a essa variável. Da informação recolhida consta o nome da classe, o GlobalRef respeitante à variável partilhada, a expressão a atribuir à variável partilhada e as variáveis que fazem parte dessa expressão. Estas informações levarão à geração duma classe de fecho, respeitante à atribuição da expressão à variável partilhada guardada no GlobalRef, no seu *place* de origem.

O nó respeitante ao LocalAssign é então modificado para um Call, que invoca a operação para mudar a execução do programa para o *place* de origem do GlobalRef.

Local Este nó da AST refere-se a uma dada variável que já foi declarada anteriormente. É necessário analisar estes nós nos casos em que representam variáveis distribuídas ou partilhadas. Nesses casos procede-se à renomeação das variáveis em causa, para facilmente diferenciar no código as variáveis qualificadas com os modificadores **dist** ou **shared** pelo programador.

Special Este nó da AST é relativo às variáveis especiais this e super.

No caso do this este é substituído pela variável object$class. Este vai ser o nome da variável que vai conter o objecto da classe que alberga o método SOMD.

No caso do super é realizada a substituição deste por um *cast* para a superclasse da variável object$class.

Barrier Serve apenas para indicar se o método em questão faz uso de barreiras ou não. Caso faça esta informação influenciará a criação de classes no passo SpringX10SOMDPrettyPrinter.

Return O retorno do método SOMD é mapeado na chamada de função que guardará o resultado a retornar num vector partilhado array$results caso se encontre num ambiente de memória partilhada, ou no vector armazenado no GlobalRef array$resultsdist caso se use memória partilhada distribuída. O resultado é guardado no índice resultante da expressão $here\$place \times NTHREADS + thread\$t$. As variáveis here$place e thread$t dizem respeito ao número do *place* onde a computação se encontra e número da *thread* dentro dum dado *place*, respectivamente.

Call Este nó refere-se às chamadas de métodos existentes no código. A transformação realizada neste ponto envolve a substituição duma chamada a um método SOMD pelo construtor da classe que representará esse método, sendo realizada de seguida a invocação do método que se pretende usar.

Isto permite assim a chamada de outros métodos SOMD dentro dum método SOMD.

4.5.3 Passo SpringX10SOMDPrettyPrinter

O último passo adicionado ao processo de compilação diz respeito à criação das classes Java necessárias ao programa, através do uso de *templates* de código que são completados com as informações recolhidas no passo anterior.

O resultado final deste passo são ficheiros Java que contêm classes que permitem a execução do modelo SOMD e as classes Java presentes nos ficheiros de entrada. De seguida o compilador termina a sua função compilando estes ficheiros para Java *bytecode*.

O uso de *templates* para gerar as classes prendeu-se com o facto destas serem bastante extensas, cerca de centenas linhas de código. A sua manipulação ao nível de nós AST requereria a criação de um muito elevado número de nós, para criar o mesmo código presente nos *templates*. Assim os *templates* de código foram a solução adoptada, tendo sido criados a partir de programas X10 que foram compilados para a linguagem Java, usando as opções de compilação -O e -NO_CHECKS do compilador do X10. Estas opções dizem respeito ao uso de optimizações no código e à desactivação da verificação dos limites de vectores, apontadores nulos e se o conteúdo dos DistArrays está contido no *place* em questão, respectivamente.

Cada método SOMD origina a criação duma classe construída a partir dos *templates*. Cada *template* apresenta código Java com etiquetas, que começam com o símbolo # seguidas dum nome e.g. #CLASSNAME, onde devem ser inseridas informações referentes a nomes de variáveis, ou nomes de classes, distribuições, classes de fecho, aplicar uma dada redução, entre outras. Entre os *templates* criados encontram-se:

- SOMDClass;
- CreateShared;
- GetSharedClosure;
- SetSharedClosure;
- SetResultClosure;
- DivideArrayPerPlace;
- DivideArrayPerThread;
- AtEachPlaceClosure;
- ExecParallelClosure;

- ApplyReduction.

Segue-se a descrição detalhada de cada uma, podendo os *templates* ser consultados no anexo A.

SOMDClass Este *template* gera a classe que mapeia um método SOMD no modelo de execução com o mesmo nome. A classe presente no *template* extende a classe a SOMDEnqueuer, sendo o tipo desta classe preenchido com o tipo de retorno do método SOMD orignal.

Os membros de dados que este apresenta dizem respeito ao sistema de serialização da biblioteca X10, contudo se o método SOMD apresentar parâmetros, estes farão parte também dos membros de dados da classe, sendo passados através do construtor da classe. Caso o método SOMD não seja void existirá uma variável que guardará o resultado da computação do método SOMD, pertencendo esta também aos membros de dados da classe.

De seguida é preenchido o método que irá executar o método segundo o modelo SOMD. A listagem 4.6 apresenta a estrutura deste método mostrando também as etiquetas que possui. As etiquetas presentes nesta listagem são:

- #RETURNTYPE - tipo de retorno do método;
- #METHODNAME - nome do método;
- #PARAMETERS - parâmetros dos método;
- #TRANSFORMARRAYSTODIST - cria as partições dos dados usando as distribuições definidas pelo programador e cria as variáveis GlobalRef que mapeiam as variáveis partilhadas e a variável onde são guardados os resultados das computações;
- #APPLYREDUCTION - aplica o *template* relativo à redução;

Relativamente ao lançamento da execução paralela do programa, este refere-se ao lançamento das *actividades* do sistema de execução X10, que irão executar o código relativo ao método SOMD. Existem duas formas de lançar as *actividades*, uma para sistemas *multi-core* onde são lançadas apenas actividades, e outra para sistemas distribuídos onde são lançadas execuções nos *places* existentes. Nesta fase e caso se use as distribuições por omissão, os *templates* relativos a estas são aplicados antes do lançamento de cada *actividade* ou lançamento de execução num dado *place*.

A seguir ao método são aplicados os *templates* referentes à criação das classes de fecho relativas à execução das actividades e *gets* e *sets* realizados a variáveis mapeadas em GlobalRefs (variáveis partilhadas e variável que guardará o resultados das computações das *actividades*).

Este *template* termina com as definições das implementações dos métodos herdados da interface SOMD.

Listagem 4.6: Estrutura resumida método principal do *template* SOMDClass

```
public #RETURNTYPE #METHODNAME(#PARAMETERS) {
  #TRANSFORMARRAYSTODIST
  ...
  // lancamento da execucao paralela do programa
  ...
  #ADDRESULTTEMPLATE
}
```

CreateShared Todas as operações necessárias à criação duma variável GlobalRef a partir do código X10 compilado para Java encontram-se neste *template*. As informações necessárias para criar uma variável deste género são: o tipo da variável **shared**, o nome dessa variável e o seu valor inicial, caso tenha.

No caso de ser dado um valor inicial à variável **shared**, a declaração do GlobalRef é seguida da atribuição do valor dado ao mesmo.

GetSharedClosure, SetSharedClosure e SetResultClosure Estes *templates* dizem respeito à criação das classes de fecho necessárias para mapear as operações leitura e escrita sobre GlobalRefs. Num sistema composto por um único *place* (uma só JVM) as operações sobre os GlobalRef podem ser executadas localmente. No entanto, num sistema distribuído é necessário encaminhar as operações sobre este tipo de variáveis para o *place* de origem.

O *template GetSharedClosure* cria a classe de fecho que acede e retorna o valor de um dado GlobalRef. Assim a informação essencial a este *template* é o GlobalRef referente à variável partilhada que se quer aceder e o tipo de retorno da variável partilhada.

Já os *templates SetSharedClosure* e o *SetResultClosure* são referentes à atribuição de valores à variável contida no GlobalRef. A diferença entre estes prende-se na especialização do *template SetResultClosure*, na qual não é necessária a informação sobre a atomicidade da afectação do resultado (uma vez que cada *actividade* escreve numa posição específica do vector), presente no *template SetSharedClosure*.

As informações necessárias ao preenchimento do *template SetSharedClosure* são: o GlobalRef da variável partilhada, as variáveis que participam na expressão a atribuir à variável **shared** e a própria expressão que será atribuída. No caso do *SetResultClosure* a informação é análoga exceptuando o facto de não haver expressão neste caso, sendo neste caso apenas necessário o tipo da variável a guardar no vector.

DivideArrayPerPlace e DivideArrayPerThread Estes *templates* referem-se à criação de partições de dados aplicando à distribuição por omissão a vectores, sendo o *DivideArrayPerPlace* respeitante à partição dos dados por *places* e *DivideArrayPerThread* referente à partição dos dados por *actividades*. A fórmula utilizada para realizar a partição dos dados ao nível dos *places* e ao nível das *actividades* é a mesma, a menos do uso de variáveis referentes a cada um desses níveis. O resultado desta partição é uma distribuição equitativa

dos dados tanto quanto possível, a estratégia neste caso passa por dividir o vector pelo número de trabalhadores, sendo o resto da divisão distribuído pelas partições.

A listagem 4.7 apresenta o código relativo à fórmula usada para particionar um *array* por *places*. Esta fórmula é aplicada à iteração de *places* sendo as etiquetas relativas a #ARRAY, #DISTARRAY e #TYPE referentes a vector a particionar, nome da variável correspondente à partição dos dados e tipo da estrutura dados, respectivamente. A fórmula apresentada é análoga para particionar dentro dum *place* por *actividades*. Nesta fórmula a expressão place.id corresponde a um identificador inteiro atribuído a cada *place*, sendo que no caso das *actividades* este é substituído pelo número identificador da *actividade* num dado *place*.

Listagem 4.7: Excerto de código que contém a fórmula usada para definir a distribuição por omissão

```
int regionSize = #ARRAY.length;
int numPlaces = x10.lang.Place.numPlaces$O();
int perPlace = regionSize / numPlaces;
int remainder = regionSize % numPlaces - place.id > 0 ? 1 : 0;
int offset = java.lang.Math.min(place.id, regionSize % numPlaces);
int begin = place.id * perPlace + offset;
int end = begin + perPlace + remainder - 1;

#TYPE #DISTARRAY = java.util.Arrays.copyOfRange(#ARRAY, begin, end+1);
```

AtEachPlaceClosure O *template* define a classe AtEachPlaceClosure que se insere nos trabalhos de extensão deste modelo de execução a ambientes de execução distribuídos. A classe distribui as partições de dados atribuídas a cada *place* pelas *actividades* que irão executar nesses mesmos *places* e lança a sua execução. As informações necessárias para preencher este *template* são os parâmetros e variáveis locais SOMD, imprescindíveis para executar o método.

ExecParallelClosure O *template* define a classe ExecParallelClosure que encapsula as acções a executar por cada instância do método, correspondendo ao código que uma *actividade* irá executar. O código relativo ao corpo método, é copiado para o método apply presente nesta classe. Para preencher a restante informação necessária pelo *template* é necessário fornecer a informação sobre os parâmetros e variáveis locais.

ApplyReduction Diz respeito ao código relativo à redução do método SOMD. Este *template* é aplicado à classe produzida pelo *template SOMDClass*, sendo o código referente à obtenção do vector de resultados armazenado no vector partilhado array$results, ou no GlobalRef array$resultsdist dependendo do ambiente de memória utilizado, e à chamada do método *reduce* do construtor de classe passado ao **reducewith**. Terminando este *template* com o retorno do resultado computado no *reduce*.

Transformações finais

Os nós de AST referentes aos construtores SOMD estão até a esta altura inalterados. Para finalizar a geração do código é necessário proceder à sua transformação final para que o compilador Java possa reconhecer o código e gerar o Java *bytecode*. As alterações efectuadas encontram-se descritas na tabela 4.4.

Tabela 4.4: Transformações realizadas aos nós de AST

AtomicBlock	`try {`
	`  x10.lang.Runtime.enterAtomic();`
	`  statements`
	`} finally {`
	`  x10.lang.Runtime.exitAtomic();`
	`}`
	Onde *statements* são operações realizadas dentro do bloco **atomic**

Barrier	`clock.advance();`
DistParameter	Perde o qualificador **dist**
DistVar	À semelhança do DistParameter também perde o qualificador **dist**

SOMDBlock	`XRT.SOMDRuntime.waitInit();`
	`final SOMDEnqueuer<#TYPE> somd = new`
	`  #SOMDClass((System[])null).init(#PARAMETERS);`
	`XRT.SOMDRuntime.enqueue(somd);`
	`[#TYPE result =] somd.getResult();`
	`[return result;]`
	Onde #TYPE refere-se ao tipo de retorno do método, #SOMDClass ao nome da classe SOMD gerada com a partir dos *templates* e #PARAMETERS aos parâmetros do método. Caso o tipo de retorno seja diferente de *void* então o resultado do método *getResult* é guardado numa variável, sendo efectuado o retorno da mesma.

SharedVar	Tal como as variáveis distribuídas perde o seu qualificador, **shared**
SpringMethodDecl	A informação relativa ao construtor **reducewith** é retirada

Terminada a explicação de como foi implementado este protótipo, com a criação do compilador que reconhece a extensão à linguagem Java, no próximo capítulo será discutida avaliação realizada ao protótipo criado.

5

Avaliação

Este capítulo apresenta uma avaliação de desempenho do protótipo implementado. Foram realizadas duas análise: uma que foca os ganhos de desempenho (*speedup*) obtidos relativamente a versões sequenciais dos métodos Java e com versões sequenciais dos métodos das implementações SOMD; e uma segunda que foca a qualidade do protótipo quando comparado com programas de *benchmark* da linguagem X10.

Além disso foi efectuada uma pequena análise de produtividade, baseada no número de linhas necessárias à implementação das políticas de distribuição e redução usadas pelos métodos. Todas as aplicações usadas nesta avaliação estarão disponíveis em `http://asc.di.fct.unl.pt/serco/somd`.

5.1 Análise de desempenho

A análise efectuada envolveu a implementação métodos SOMD para aplicações estatísticas, aritméticas e ainda algoritmos de ordenação. Sendo esse conjunto bastante vasto para ser apresentado na sua totalidade, optou-se por escolher implementações representativas dos métodos testados: cálculo dum *histograma*, *média* dos elementos dum vector, o algoritmo de ordenação *Merge Sort*, a *soma* de dois vectores, o algoritmo de *K-means* e duas implementações de *multiplicação de matrizes*: *linhas* (MML) que distribui as linhas da primeira matriz e *colunas* (MMC) que distribui as colunas da segunda.

Nestes testes foram utilizados diversos tamanhos para os dados de entrada de ordens de magnitude diferentes, tendo sido classificadas em classes de problemas de A a D. As medições de tempos foram realizadas recolhendo várias medições para cada teste, em que se retirou cerca de 15% melhores e dos piores tempos, tendo-se efectuado a média das restantes medições. A tabela 5.1 apresenta a configuração dos dados de entrada para

cada método, o tempo de execução da versão sequencial Java de cada método, que apenas contabiliza o tempo da invocação e execução do método SOMD (tempo de execução) e o custo induzido pela partição dos dados para cada classe de problemas, quando o método é executado pelo sistema de execução X10 (custo da partição). De referir ainda que o tamanho do conjunto de dados de usado no *K-means* era composto por *observações* × *atributos*, sendo o número de atributos igual a 4 para todas as classes de problemas.

As medições de tempos foram efectuadas num sistema composto por dois CPU Quad-Core AMD Opteron a 2.3GHz, com 16 Gigabytes de memória RAM, a executar a versão 2.6.26-2 do sistema de operação Linux.

Tabela 5.1: Tabela de referência com as configurações de cada classe

	Classe A			Classe B		
Aplicações	Configuração	Execução tempo (s)	Partição custo (s)	Configuração	Execução tempo (s)	Partição custo (s)
Histograma	tamanho do vector: 100.000.000	2.34	0.5	tamanho do vector: 10.000.000	0.22	0.055
Média	tamanho do vector: 100.000.000	0.21	0.5	tamanho do vector: 10.000.000	0.02	0.055
Soma	tamanho do vector: 100.000.000	0.20	1.02	tamanho do vector: 10.000.000	0.02	0.1
Merge Sort	tamanho do vector: 100.000.000	23.98	0.5	tamanho do vector: 10.000.000	2.18	0.065
K-means	nr observações: 25.000.000	141.47	0.165	nr observações: 2.500.000	14.26	0.017
MML	tamanho da matriz: 2000×2000	222.94	0.00007	tamanho da matriz: 1500×1500	82.87	0.00005
MMC	tamanho da matriz: 2000×2000	222.94	0.05	tamanho da matriz: 1500×1500	82.87	0.04
	Classe C			Classe D		
Histograma	tamanho do vector: 1.000.000	0.024	0.0045	tamanho do vector: 100.000	0.006	0.00045
Média	tamanho do vector: 1.000.000	0.005	0.0045	tamanho do vector: 100.000	0.003	0.00045
Soma	tamanho do vector: 1.000.000	0.006	0.008	tamanho do vector: 100.000	0.005	0.0008
Merge-sort	tamanho do vector: 1.000.000	0.231	0.0045	tamanho do vector: 100.000	0.061	0.00045
K-means	nr observações: 250.000	1.40	0.0015	nr observações: 25.000	0.185	0.0002
MML	tamanho da matriz: 750×750	3.35	0.00004	tamanho da matriz: 250×250	0.148	0.000035
MMC	tamanho da matriz: 750×750	3.35	0.033	tamanho da matriz: 250×250	0.148	0.005

As tabelas de 5.2 até 5.5 apresentam as medições efectuadas para as aplicações com implementação SOMD, respeitantes a cada classe. Estas medições contam com o tempo de execução do método assim como o tempo na fila de trabalhos. Contudo este último é negligenciável, uma vez que na execução dos testes era garantido que apenas um método era executado de cada vez.

Os gráficos das figuras 5.1a a 5.1d apresentam os *speedups* obtidos pelas implementações SOMD das aplicações, quando comparadas com os métodos Java originais. Esta

Tabela 5.2: Medições dos problemas da classe A

	Tempo de execução (s) - Classe A						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	2.8354	0.7028	2.0597	31.2078	138.3764	222.6412	222.7932
2	1.6795	0.5931	1.8277	17.1471	62.9661	111.3317	79.0965
4	1.0981	0.5528	1.7356	11.1054	36.2014	56.1223	27.5471
7	0.8750	0.5592	1.7425	9.6015	23.8448	32.3031	16.6955
8	0.8709	0.5597	1.8746	10.5098	23.9040	28.2747	15.2686

Tabela 5.3: Medições dos problemas da classe B

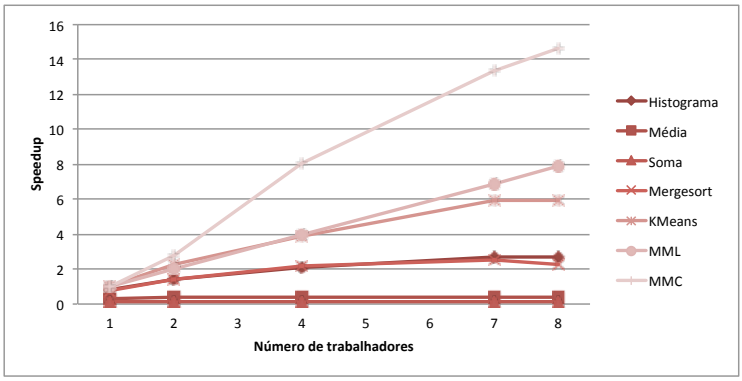
	Tempo de execução (s) - Classe B						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	0.2794	0.0810	0.2271	2.9177	14.0412	84.6538	82.8085
2	0.1805	0.0800	0.3099	1.7388	6.7195	41.9175	23.2265
4	0.1532	0.0780	0.2526	1.1787	3.6194	20.7317	7.3079
7	0.2336	0.1703	0.3609	1.1099	5.8782	12.1039	4.3459
8	0.2311	0.1900	0.3930	1.3617	7.1803	10.7006	3.9792

Tabela 5.4: Medições dos problemas da classe C

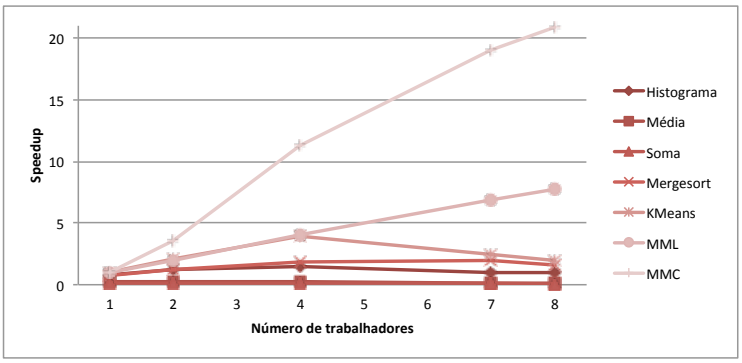
	Tempo de execução (s) - Classe C						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	0.0357	0.0997	0.1027	0.3060	1.5201	3.3372	3.4375
2	0.0235	0.0585	0.1028	0.2237	0.8393	1.7375	1.6854
4	0.1088	0.0209	0.1781	0.2099	0.5910	0.9755	0.9138
7	0.0280	0.0245	0.0553	0.2270	1.2109	0.6334	0.6095
8	0.0391	0.0256	0.0558	0.1885	1.4772	0.6285	0.6660

Tabela 5.5: Medições dos problemas da classe D

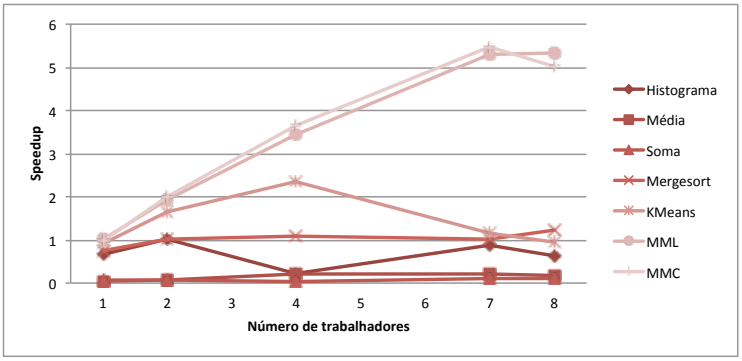
	Tempo de execução (s) - Classe A						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	0.0117	0.0090	0.0180	0.0594	0.2609	0.1453	0.1608
2	0.0147	0.0111	0.0200	0.0665	0.2216	0.0897	0.0638
4	0.0150	0.0137	0.0228	0.0602	0.3603	0.0744	0.0611
7	0.0217	0.0240	0.0235	0.1495	0.6358	0.0838	0.0645
8	0.0278	0.0340	0.1542	0.2618	0.6482	0.0868	0.0851



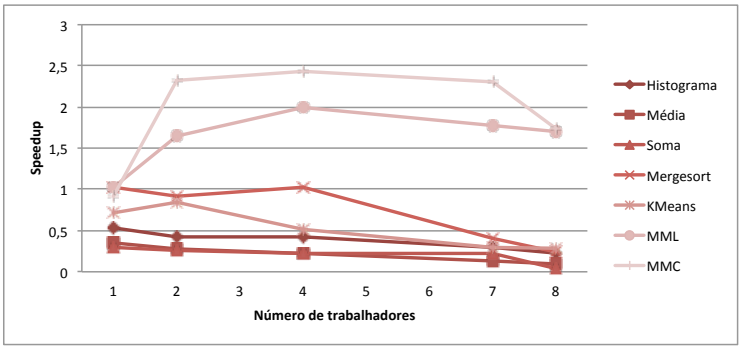
(a) Classe A



(b) Classe B



(c) Classe C



(d) Classe D

Figura 5.1: *Speedup* - Java (retirado de [MP12])

comparação tem como objectivo observar o comportamento da execução paralela usando a aplicação directa do paradigma DMR, quando comparado com o método original. Aplicação directa indica que o método original não foi alterado de modo a que o paradigma seja usado. A única aplicação cujo este paradigma não foi aplicado de forma directa é o caso do *K-means*, cujo o código foi alterado de modo a que a redução acontecesse na execução paralela do algoritmo.

Analisando os resultados observa-se que as aplicações em que foram usadas matrizes, *K-means*, *MML* e *MMC*, são as aplicações que tiveram um melhor desempenho ao nível do *speedup*. Isto pode ser explicado pelo peso computacional dessas mesmas aplicações. Destas, a multiplicação de matrizes por linhas, *MML*, atinge um *speedup* linear nas classes A e B, apresentando no seu pico 7.80 e 7.76 respectivamente, baixando o seu desempenho para 5.42 na classe C e na classe D para 1.67.

O gráfico referente à classe A desta figura mostra logo que as aplicações *Soma* e *Média* são maus candidatos de aplicações a paralelizar usando o mecanismo DMR por omissão, apresentando um *speedup* negativo.

O desempenho do *Merge Sort* encontra-se limitado pela sua fase redução, na qual é necessário reordenar os vectores parciais de modo a produzir o resultado final, sendo este um bom candidato para a implementação duma estratégia de redução paralela. Apesar de ter obtido algum *speedup* nas classes A e B, este é considerado um *speedup* tímido.

O *Histograma* à semelhança do *Merge Sort* também não é um bom candidato à paralelização, pois apresentam apenas um *speedup* tímido nas classes A e B, atingindo um *speedup* negativo na classe C. Isto deve-se ao seu peso computacional não ser suficiente para obter melhores resultados.

As aplicações que obtiveram melhores desempenhos foram o *K-means*, *MML* e *MMC*, isto pode ser explicado pelo peso computacional destas aplicações. No caso do *K-Means* apresenta *speedups* interessantes nas classes A e B atingindo os valores de 5.93 e 3.93 respectivamente. Na classe C ainda apresenta um *speedup* quando usados quatro trabalhadores, contudo já não se consegue obter esse desempenho na classe D.

O *MML* apresenta um *speedup* linear com picos de 7.87 na classe A e 7.74 na classe B. Na classe C apresenta já um ganho menor atingindo um *speedup* máximo de 5.3. Porém na classe D o ganho no desempenho já é bastante menor sendo o valor máximo de 1.99 com 4 trabalhadores. A multiplicação de matrizes por colunas, *MMC*, por sua vez apresenta um *speedup* super-linear nas classes A e B com valores máximos de 14.59 e 20.82 respectivamente, beneficiando assim dum mapeamento na memória que vai de encontro hierarquia de memória do computador. Nas classe C e D o seu desempenho desce consideravelmente apresentando valores já semelhantes ao seu congénere *MML*. De referir que o comportamento desta multiplicação na classe D apresenta resultados um pouco melhores que a MML apresentando um *speedup* máximo de 2.63 nesta classe.

Para o próximo grupo de medições foi aplicada uma estratégia em que se tentou reduzir o custo de criar as partições. Este custo foi reduzido definindo intervalos para as estruturas de dados, invés de se particionar as próprias estruturas. Isto deu origem

a modificações no código das aplicações introduzindo a noção de computação parcial, uma vez que se deixa a ilusão de se estar a trabalhar sobre a estrutura completa. Esta estratégia será referida a partir daqui como solução baseada em intervalos. A listagem 5.1 mostra a implementação do método *Soma* usando esta estratégia. As tabelas 5.6 a 5.9 são relativas às medições efectuadas neste contexto.

Listagem 5.1: Implementação baseada em intervalos do método *Soma*

```
1 void sum(int[] array1, int[] array2, dist int[] range) {
2   int begin = range[0];
3   int end = range[1] + 1;
4   for ( int i = 0; i < (end - begin); i++)
5     array1[i + begin] = array1 [i + begin] + array2 [i +begin] ;
6 }
```

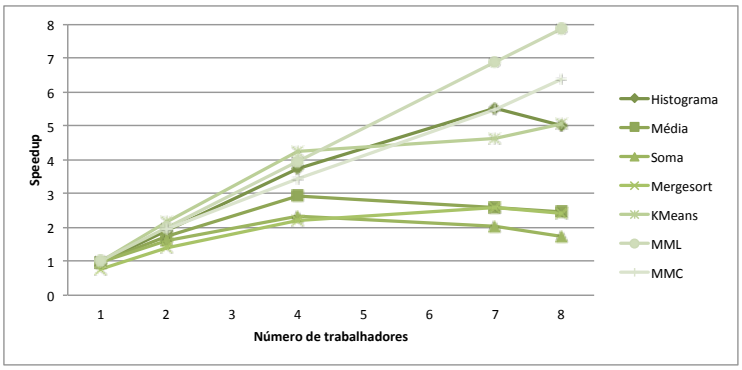
Tabela 5.6: Medições das implementações com intervalos para a Classe A

	Tempo de execução (s) - Classe A						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	2.3994	0.2164	0.2062	31.1447	142.1173	223.5662	230,1234
2	1.2320	0.1185	0.1236	17.0335	65.4200	111.4784	114,9744
4	0.6275	0.0712	0.0852	10.8190	33.2723	56.0799	64,9544
7	0.4226	0.0795	0.0988	9.2904	30.4867	32.3454	40,5015
8	0.4638	0.0844	0.1149	9.8711	27.8726	28.3949	34,9357

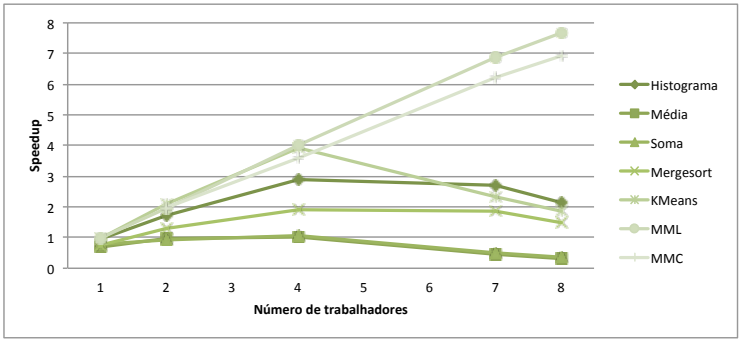
Tabela 5.7: Medições das implementações com intervalos para a Classe B

	Tempo de execução (s) - Classe B						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	0.2355	0.0318	0.0312	2.9075	15.0184	84.4511	87.0371
2	0.1302	0.0233	0.0264	1.6508	6.8275	41.9107	41.9347
4	0.0776	0.0223	0.0232	1.1405	3.6376	20.6789	23.0141
7	0.0830	0.0501	0.0506	1.1704	6.0805	12.0577	13.3625
8	0.1030	0.0739	0.0685	1.4523	7.6135	10.7712	11.9435

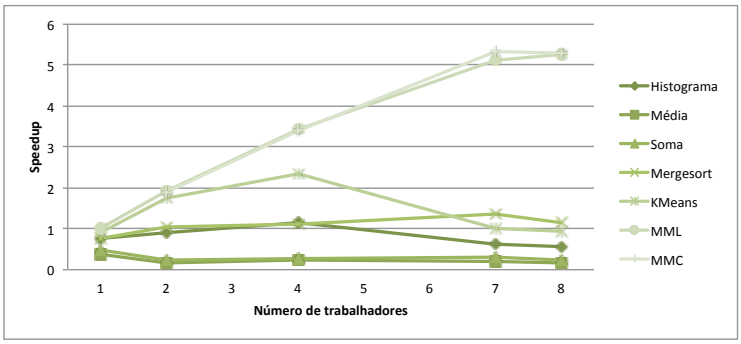
Os gráficos da figura 5.2a a 5.2d dizem respeito ao cálculo do *speedup* das aplicações usando a estratégia de distribuição com intervalos, quando comparadas com a versão original do método Java. Observando o gráfico relativo à classe A, verificamos que todas as aplicações conseguiram *speedup* positivo, o que não se sucedia com a estratégia anterior. O *speedup* obtido foi bastante generoso para quase todas aplicações, exceptuando a *Média*, a *Soma* e o *Merge Sort*.



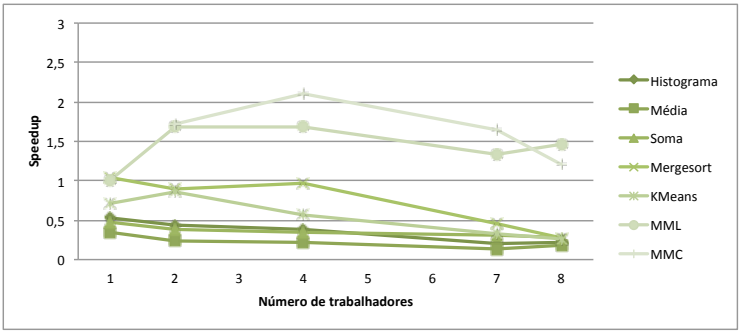
(a) Classe A



(b) Classe B



(c) Classe C



(d) Classe D

Figura 5.2: *Speedup* - Java usando intervalos (retirado de [MP12])

Tabela 5.8: Medições das implementações com intervalos para a Classe C

	Tempo de execução (s) - Classe C						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	0.0317	0.0129	0.0128	0.3045	1.5423	3.3357	3.4123
2	0.0262	0.0269	0.0281	0.2223	0.7956	1.7386	1.7679
4	0.0210	0.0196	0.0243	0.2088	0.5992	0.9702	0.9847
7	0.0396	0.0225	0.0202	0.1686	1.3693	0.6548	0.6294
8	0.0427	0.0294	0.0276	0.1985	1.4662	0.6354	0.6303

Tabela 5.9: Medições das implementações com intervalos para a Classe D

	Tempo de execução (s) - Classe D						
IMs SOMD	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
1	0.0118	0.0091	0.0109	0.0591	0.2590	0.1486	0.1475
2	0.0144	0.0133	0.0135	0.0681	0.2141	0.0884	0.0859
4	0.0168	0.0149	0.0149	0.0629	0.3238	0.0882	0.0705
7	0.0310	0.0235	0.0169	0.1350	0.5783	0.1110	0.0903
8	0.0296	0.0181	0.0185	0.2223	0.7064	0.1017	0.1227

As aplicações que melhor desempenho apresentam usando esta estratégia são as multiplicações de matrizes, seguidas pelo *K-Means* e *Histograma*. Porém este último apesar de apresentar um bom resultado na classe A, começa rapidamente a perder esse efeito nas classes seguintes.

A aplicação *MML* apresenta à mesma um *speedup* linear até à classe D e a sua congénere *MMC* perdeu o efeito de *speedup* super-linear que apresentava nas classes A e B da aplicação directa do modelo DMR, dando origem a um *speedup* linear que se mantém também até à classe D. A aplicação *K-means* voltou acompanhar as multiplicações de matrizes até à classe D, tendo tido um *speedup* interessante até certos pontos nas classes anteriores, contudo na classe D voltou a mostrar um desempenho negativo.

Para analisar melhor os custos associados a cada uma das estratégias apresenta-se na tabela 5.10 as medições das estratégias implementadas para cada aplicação, na sua versão sequencial. Observamos que as estratégias adoptadas pelas implementações SOMD apresentam na sua generalidade um desempenho pior, devendo-se isto ao tempo consumido nas fases de distribuição de dados e de redução de resultados que acaba por ser penalizador na versão sequencial.

Das estratégias adoptadas nas implementações SOMD observa-se que apesar da estratégia usando intervalos se mostrar eficaz nos programas com tempos de execução mais baixos e em que se usam vectores, esta solução acaba por ter desempenho pior quando as aplicações fazem uso de matrizes.

Tabela 5.10: Medições das versões sequenciais das diferentes implementações - classe A

	Tempo de execução (s)						
Estratégia	Histograma	Média	Soma	Merge Sort	K-Means	MML	MMC
Partições	2.8354	0.7028	2.0597	31.2078	138.3764	222.6412	222.7932
Intervalos	2.3994	0.2164	0.2062	31.1147	142.1773	223.5662	230.1234
Original	2.3378	0.2075	0.2003	23.9821	141.4682	222.9378	222.9378

5.2 Comparação entre a implementação SOMD e X10

Esta segunda análise tem como objectivo aferir da qualidade do protótipo implementado. Com esse propósito foram implementados segundo o modelo SOMD, um conjunto de aplicações de *benchmark* do X10, tendo-se comparado os resultados de ambas as implementações. O código usado pelo X10 foi compilado com as *flags* -O e -NO_CHECKS que permitem uma compilação optimizada e uma execução mais rápida dos programas.

As aplicações incluídas neste estudo são:

- Monte Carlo - cálculo duma estimativa para o valor da constante π através do método de Monte Carlo;
- Mandelbrot - cálculo do conjunto de Mandelbrot;
- Crypt - cifrar e decifração dum texto;
- Stream - computação da fórmula $a = b + \alpha \times c$, onde a, b e c são vectores de inteiros e α uma constante.

Todas as aplicações, com a excepção do *Monte Carlo*, distribuem pelo menos um vector. Sendo as configurações usadas: *Crypt* distribui um vector de 50 milhões de bytes; *Mandelbrot* executa duas distribuições, uma sobre uma matriz de 6000×750 e outra sobre um vector de 750 doubles; e o *Stream* executa três distribuições sobre vectores de 33 Megabytes. Neste contexto não faz sentido aplicar a estratégia de intervalos ao código da computação de *Monte Carlo* uma vez que este já usa intervalos para fazer a sua computação. Todas as configurações indicadas fazem parte dos valores testados nos *benchmarks* do X10.

As tabelas 5.11 e 5.12 são referentes às medições das aplicações segundo as estratégias de partições e de intervalos respectivamente. A tabela 5.13 apresenta os valores obtidos pelas aplicações X10.

Na análise dos tempos da tabela destacamos o caso da implementação do *Stream* com a estratégia de partições, cujo o tempo de perda anda à volta de 18 vezes mais lento que o seu congénere X10. Isto pode ser explicado pelo número de distribuições que se realizaram e pelo tamanho dos dados a distribuir, que num programa com um tempo

de execução baixo como é o caso, influencia bastante no seu desempenho, pois as distribuições são parte significativa da computação. Este resultado é melhorado usando a estratégia dos intervalos para as distribuições.

Tabela 5.11: Medições dos *benchmarks* SOMD

	Tempo de execução (s)			
IMs SOMD	Montecarlo	Mandelbrot	Crypt	Stream
1	5.4489	29.1913	3.8304	1.9072
2	2.7506	14.8489	2.3695	1.7215
4	1.3996	12.1889	1.4974	1.6004
7	0.9630	8.3353	n/d	1.4538
8	0.9548	7.2813	1.4697	1.7211

Tabela 5.12: Medições dos *benchmarks* SOMD com intervalos

	Tempo de execução (s)		
IMs SOMD	Mandelbrot	Crypt	Stream
1	29.5575	3.4168	0.3292
2	14.8315	1.7664	0.1908
4	12.1040	0.9343	0.1317
7	8.4031	n/d	0.1385
8	7.4991	0.7429	0.1477

Tabela 5.13: Medições dos *benchmarks* X10

	Tempo de execução (s)			
IMs SOMD	Montecarlo	Mandelbrot	Crypt	Stream
1	5.7596	28.5223	5.9534	0.2538
2	2.9118	14.5619	3.1032	0.1811
4	0.9810	11.5922	1.6957	0.1451
7	0.9810	8.3441	n/d	0.1333
8	0.9977	7.6217	1.3579	0.2190

O gráfico da figura 5.3 apresenta a razão da diferença dos tempos das aplicações SOMD com as aplicações X10. Na legenda as aplicações terminadas com um I no final dizem respeito às aplicações que usaram intervalos. Deste gráfico foi omitida a informação de comparação relativa ao programa *Stream* usando partições, por apresentar um valor demasiado elevado que tornaria irrelevantes as restantes medições do gráfico.

Do gráfico observamos que os *benchmark Mandelbrot* e *Monte Carlo* obtêm resultados semelhantes ao do X10, com o X10 a escalar melhor neste último e o SOMD em ambas as implementações do *Mandelbrot* a ganhar uma pequena vantagem.

Tal como se previa a versão *StreamI* consegue um desempenho bem superior que a

implementação *Stream*, uma vez que se reduz o custo de particionar os vectores, conseguindo a passos escalar melhor que o X10.

Finalmente a aplicação *Crypt* perde comparativamente para o X10, escala melhor que o SOMD até ter um ganho de 38% com oito trabalhadores, ajudando as partições dos dados a contribuir para esse tempo. Por sua vez a implementação *CryptI* apresenta resultados bem superiores ao do X10, podendo isto ser explicado pelo código gerado pelo X10, cujo o método principal executa perto de 1000 comandos Java, enquanto o mesmo método no SOMD apresenta apenas 60 desses comandos.

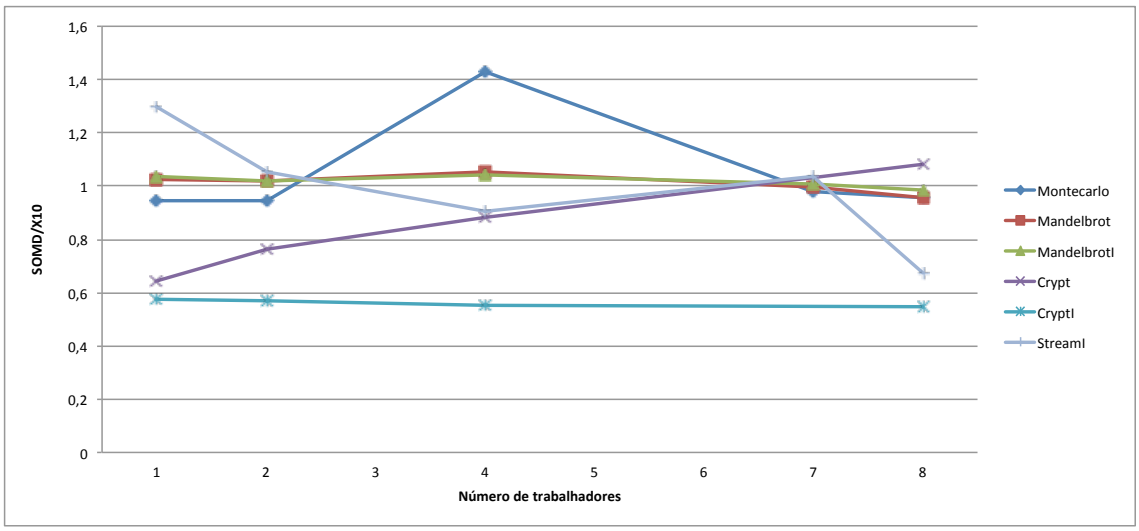


Figura 5.3: Comparação entre SOMD e X10 (retirado de [MP12])

5.3 Análise da Produtividade

Foi realizada uma pequena análise de produtividade ao medir o número de linhas de código necessárias para implementar as políticas de distribuição e redução para cada aplicação. A tabela 5.14 apresenta essas medidas, corroborando a nossa ideia de que o modelo proposto é simples de usar. Esta afirmação é ainda mais substanciada pelos factos de que:

1. a maior parte dos métodos de manipulação de vectores recorrem a esta distribuição por omissão, sendo a distribuição indicada pelo programador usada em casos mais especializados;

2. a maior parte das distribuições e reduções são gerais o suficiente para se encontrarem sob a forma duma biblioteca;

3. as computações que executam problemas de algoritmia não necessitam de conhecimento especial sobre programação paralela, ficando a ressalva de que nem todas as implementações poderão gerar programas eficientes.

Tabela 5.14: Linhas de código das distribuições e reduções nas aplicações que usam a estratégia de partições

Aplicações	Distribuições	Reduções	Total
Histograma	Por omissão	8	8
Média	Por omissão	6	6
Merge Sort	Por omissão	40	40
K-means	Por omissão	8	8
Multiplicação de matrizes - linhas	Por omissão	14	14
Multiplicação de matrizes - colunas	19	12	31

Tabela 5.15: Linhas de código dos *benchmarks*

Benchmarks	SOMD	X10
Crypt	208 + 15	202
Mandelbrot	68 + 16	83
Stream	28 + 16	33
Montecarlo	31 + 10	37

Estes resultados poderiam ainda ser melhorados, arranjando um novo construtor que usasse o mesmo código da execução na redução, uma vez que a maior parte dos métodos testados utilizam uma operação de redução igual ao código da sua execução. Esta solução melhoraria por exemplo, o resultado do *Merge Sort* no qual se implementou novamente o algoritmo de Merge Sort para aplicar a redução.

Foi ainda realizada a comparação do número de linhas dos *benchmarks* X10 quando comparados com os respectivas implementações SOMD. A tabela 5.15 apresenta para o SOMD, o número de linhas de código do programa, mais o número de linhas da redução usad. As implementações X10 apresentam sempre menos linhas de código se contarmos com a totalidade dos programas, porém se contarmos apenas com o código do programa, não contabilizando as reduções que podem surgir sob a forma de bibliotecas, as implementações SOMD apresentam na sua generalidade menos linhas de código, exceptuando o caso do Crypt.

Terminado este capítulo de avaliação segue-se o capítulo de conclusões e de informações sobre o trabalho futuro.

6

Conclusões e Trabalho Futuro

Neste capítulo faz-se o balanço dos objectivos cumpridos, avalia-se os resultados obtidos e enquadra-se o trabalho futuro a realizar no contexto desta dissertação.

6.1 Conclusões

O trabalho elaborado no contexto desta dissertação consistiu na instanciação de um modelo de execução paralelo, SOMD, na linguagem de programação Java. O objectivo foi oferecer ao programador sem experiência na área da programação paralela, meios para criar aplicações que usufruam de paralelismo de dados sem ter de criar código especializado para estas.

A sintaxe concreta apresenta ao programador um paradigma DMR em que métodos Java podem ser anotados com as políticas de distribuição e de redução a aplicar. Para tal foram definidos e implementados vários construtores que permitem expressar políticas de distribuição e redução, partilhas de zonas de memória entre as várias instâncias, atomicidade ao nível das operações e pontos de sincronização. Dos construtores sugeridos apenas o **distshared** não se encontra devidamente implementado por questões de desempenho, tendo sido apenas criado o protótipo que simula o seu comportamento.

Verificou-se que a grande maioria dos casos de estudo adoptados, a distribuição por omissão é indicada para realizar o particionamento dos dados dos vectores. No entanto, tal não impede a implementação de estratégias específicas. Um exemplo de tal abordagem foi a aplicação de uma distribuição por colunas na multiplicação de matrizes.

Quando comparado com os mecanismos existentes no X10, a classe que suporta a distribuição automática dos dados, DistArray, não permite a criação de distribuições específicas para estruturas de dados que não sejam vectores. Porém isto pode ser conseguido no

X10 através da criação das partições por parte do programador e na distribuição destas pelos *places*, contudo isto envolve trabalho do programador que se quer evitar.

Outro factor que pensamos que contribui para a redução da complexidade da aplicação do modelo é a "reusabilidade"das políticas de distribuição e redução. A quase totalidade das políticas implementadas pode ser aplicada em variados cenários, um exemplo é a aglomeração de vectores a partir dos vectores parciais. Tal permite que estas políticas possam ser apresentadas em bibliotecas, reduzindo o trabalho do programador à selecção das mesmas. Claro que tal ainda obriga o programador a saber que par distribuição/redução é o mais indicado para o seu algoritmo, mas o nível de abstracção é substancialmente superior à decomposição explícita do domínio do problema e à sua adjudicação a fluxos de execução.

Para programadores mais experientes e com conhecimentos da área é permitida a criação de programas conscientemente paralelos, cuja a implementação apresenta construtores habitualmente existentes em linguagens de computação paralela. Tal permite um grau de optimização das aplicações, tendo a noção de que estas irão ser executadas em ambientes concorrentes.

O protótipo implementado foi alvo de uma avaliação de desempenho, que pretendeu analisar em que cenários a abordagem proposta pode obter ganhos a nível do *speedup*. Nesta avaliação de desempenho estávamos interessados em aplicar este modelo de execução a operações comuns usadas no desenvolvimento de *software* e não a problemas específicos de computação paralela. Foi também efectuada uma avaliação de desempenho comparativa com o X10, com o intuito de aferir da qualidade da implementação realizada. Relativamente à primeira fase da avaliação esta apresentou resultados satisfatórios para classes de problemas com dados de entrada de grande dimensão, havendo geralmente sempre algum ganho no desempenho quando se paralelizava as aplicações. Estes ganhos foram resultado da simples anotação da distribuição e redução a usar pelas aplicações, o que perspectiva que implementações mais especializadas consigam obter um desempenho ainda melhor. Contudo as restantes classes de problemas não apresentaram os mesmos resultados, uma vez que o seu tempo de execução é demasiado baixo, para que o tempo de criação das partições não tenha impacto.

Foi ainda possível aferir que a criação de partições cria um custo acrescido demasiado elevado para classes de aplicações com tempos de execução baixos. Esse custo deve-se à criação dos vectores que contêm as partições dos dados distribuídos, que envolvem a cópia dos valores do vector inicial para os mesmos. A sua eliminação requer a adopção de estratégias como a distribuição de intervalos (ver secção 5.1) que diminuem o nível de abstracção do programador. Em linguagens de programação como o C (e derivados) poderia ser utilizada a aritmética de apontadores para reduzir este custo, através da criação das partições com o uso de apontadores. Estes indicariam onde começa cada uma das partições de dados a utilizar.

A utilização do sistema de execução do X10 provou ser bastante útil para a prova do conceito SOMD. No entanto, este impõe custos acrescidos, que apesar de serem bastante

úteis em ambientes de memória distribuída, impedem que se tenha melhores resultados. Pensamos que a implementação deste modelo, num sistema de execução dedicado beneficiaria na sua execução numa única JVM, encontrando-se trabalhos em curso para que tal se suceda.

A comparação com o sistema de execução X10 atestou a qualidade do protótipo que apresentou resultados semelhantes ao X10 na execução de um conjunto de aplicações *benchmark*. A nível de produtividade deste modelo, podemos constatar que a definição da estratégia de redução é uma constante na maior parte dos casos, sendo o programador responsável pela sua escolha ou implementação. No caso da implementação duma estratégia de redução, os valores apresentados na tabela 5.14 são indicadores que, no caso geral, a complexidade de programação destas políticas é relativamente baixa. O caso mais complexo necessitou de 40 linhas de código, pois a ordenação total dum vector implica a junção ordenada dos vectores parciais.

O trabalho realizado nesta dissertação originou já um artigo [MP12] que irá ser apresentado na conferência HPCC.

6.2 Trabalho Futuro

O trabalho realizado no âmbito desta dissertação é apenas um primeiro passo na definição e maturação do modelo SOMD. Antevê-se que o trabalho futuro incida sobre os seguintes pontos:

- Realização de testes de usabilidade dos construtores propostos com o intuito de comparar a sua facilidade de programação e expressividade com outras propostas existentes. Para estes testes será necessário um número elevado de programadores, com e sem experiência na área da programação paralela, que serão submetidos a vários exercícios nos quais utilizarão as soluções existentes e o nosso protótipo, para criar aplicações paralelas sendo sujeitos a questionário no final.

- Formalização dos construtores sugeridos, definindo as regras de semântica a que os construtores obedecem. Este ponto de trabalho já se encontra em curso.

- Implementação do modelo SOMD em outros sistemas de execução, incluindo um sistema de execução dedicado. Isto tem como objectivo comparar resultados das implementações realizadas, de modo a aferir qual a implementação que melhor desempenho apresenta para este modelo de execução. Em trabalho em decurso existe um *middleware* para programação de *clusters* de *multi-cores* [Sar12] que já incorpora este conceito.

- Paralelização dos estágios de distribuição e redução. Esta adição permitiria tirar ainda melhor desempenho das aplicações SOMD, pois neste momento a execução

destas fases ocorre de forma sequencial. No caso da distribuição paralela esta poderia ocorrer quando conjunto de dados de entrada é de grande dimensão, sendo criadas várias tarefas que calculariam as partições dos mesmos. Relativamente à fase de redução realizou-se testes que envolvem a utilização de métodos SOMD para efectuar a redução, tendo-se conseguido melhorar o desempenho do programa, quando se usou a redução em paralelo. Esta funcionalidade não se encontra implementada, pois necessita de mais trabalho a nível do compilador, que envolve a identificação das classes no estágio de redução, que no caso de serem outras classes SOMD deverão gerar código diferente nesta fase.

- Estudo para adição de novos construtores, como por exemplo a criação dum construtor para a redução que permita usar o código da execução para aplicar a redução. Este estudo teria por base a criação de vários programas de modo a abstrair comportamentos que poderiam ser abstraídos como construtores.

Bibliografia

[Agh86] Gul A. Agha. *Actors: A Model Of Concurrent Computation In Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pág. 483–485, New York, NY, USA, 1967. ACM.

[BA06] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, second edition, 2006.

[Bat80] Kenneth E. Batcher. Design of a Massively Parallel Processor. *IEEE Trans. Computers*, 29(9):837–840, 1980.

[BBNY06] Christian Bell, Dan Bonachea, Rajesh Nishtala, e Katherine Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pág. 84–84, Washington, DC, USA, 2006. IEEE Computer Society.

[BG97] Aart J. C. Bik e Dennis Gannon. Automatically Exploiting Implicit Parallelism in Java. *Concurrency - Practice and Experience*, 9(6):579–619, 1997.

[Bra00] Bradford L. Chamberlain and Sung-Eun Choi and E. Christopher Lewis and Calvin Lin and Lawrence Snyder and W. Derrick Weathersby. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Transactions on Software Engineering*, 26:2000, 2000.

[Bre09] Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, first edition, 2009.

[CCL+98] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, e Calvin Lin. The Case for High-Level Parallel Programming in ZPL. *IEEE Comput. Sci. Eng.*, 5:76–86, July 1998.

[CCZ07] Brad Lee Chamberlain, David R. Callahan, e Hans P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, e Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In Ralph E. Johnson e Richard P. Gabriel, editores, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pág. 519–538. ACM, 2005.

[CM08] Gilberto Contreras e Margaret Martonosi. Characterizing and improving the performance of Intel Threading Building Blocks. In David Christie, Alan Lee, Onur Mutlu, e Benjamin G. Zorn, editores, *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pág. 57–66. IEEE, 2008.

[CWY91] Vítor Santos Costa, David H. D. Warren, e Rong Yang. Andorra I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. *SIGPLAN Not.*, 26(7):83–93, Abril 1991.

[DBK+96] Paul Dechering, Leo Breebaart, Frits Kuijlman, Kees van Reeuwijk, e Henk Sips. A Generalized forall Concept for Parallel Languages. In *LCPC*, pág. 605–607, 1996.

[DGNP88] Frederica Darema, David A. George, V. Alan Norton, e Gregory F. Pfister. A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[DM98] Leonardo Dagum e Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[FKH+06] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, e Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[FLR98] Matteo Frigo, Charles E. Leiserson, e Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *In Proceedings of the SIGPLAN*

'98 Conference on Program Language Design and Implementation, pág. 212–223, 1998.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[GLT99] William Gropp, Ewing Lusk, e Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[GSC+10] Anwar Ghuloum, Amanda Sharp, Noah Clemons, Stefanus Du Toit, Rama Malladi, Mukesh Gangadhar, Michael McCool, e Hans Pabst. Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures. *Dr. Dobbs Go Parallel*, 210. http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=227300084.

[HCS+05] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, e Victor Basili. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pág. 35–, Washington, DC, USA, 2005. IEEE Computer Society.

[HFA99] Scott Hudson, Frank Flannery, e C. Scott Ananian. Cup LALR Parser Generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP/, 1999.

[HS08] Maurice Herlihy e Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, first edition, 2008.

[IBM11] IBM PERCS Project. *X10 Language Specifications Version 2.2*, 2011.

[KCDZ94] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, e Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter*, pág. 115–132, 1994.

[LS95] R. Greg Lavender e Douglas C. Schmidt. Active Object – An Object Behavioral Pattern for Concurrent Programming, 1995.

[LS97] Luís M. B. Lopes e Fernando M. A. Silva. Thread-and Process-based Implementations of the pSystem Parallel Programming Environment. *Software | Practice and Experience*, 27:329–351, 1997.

[MH95] John H. Merlin e Anthony J. G. Hey. An Introduction to High Performance Fortran. *Scientific Programming*, 4(2):87–113, 1995.

[MLV+03] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, e Louis Rilling. Kerrighed: A Single System Image Cluster Operating System for High Performance Computing. In Harald Kosch, László Böszörményi, e Hermann Hellwagner, editores, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pág. 1291–1294. Springer Berlin / Heidelberg, 2003.

[MP12] Eduardo Marques e Hervé Paulino. Single Operation Multiple Data - Data Parallelism at Subroutine Level. In *14th IEEE International Conference on High Performance Computing & Communications, HPCC 2012, Liverpool, United Kingdom, June 25-27, 2012*, 2012.

[Mue93] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. In *In Proceedings of the USENIX Conference*, pág. 29–41, 1993.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, e Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *In 12th International Conference on Compiler Construction*, pág. 138–152. Springer-Verlag, 2003.

[Ora11] Oracle. Remote Method Invocation Home. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`, 2011.

[PJ98] Jens Palsberg e C.Barry Jay. The Essence of the Visitor Pattern. In *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*, pág. 9 –15, aug 1998.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[SAB+10] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, e Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. Relatório técnico, Toronto, Canada, June 2010.

[Sar12] João Saramago. Um Middleware para Computação Paralela em Clusters de Multicores. Tese de Mestrado, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.

[SBS+95] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, e Charles V. Packer. Beowulf: A Parallel Workstation For Scientific Computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pág. 11–14. CRC Press, 1995.

[Sie10] Sam Siewert. Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms. `http://software.intel.com/en-us/articles/`, 2010.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, e Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[SPL99] Fernando Silva, Hervé Paulino, e Luís Lopes. di_pSystem: A Parallel Programming System for Distributed Memory Architectures. In Jack Dongarra, Emilio Luque, e Tomàs Margalef, editores, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26-29, 1999, Proceedings*, volume 1697 of *Lecture Notes in Computer Science*, pág. 525–532. Springer-Verlag, 1999.

[Sun90] Vaidy S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[THO02] Chris J. Thompson, Sahngyun Hahn, e Mark Oskin. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pág. 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[UPC05] UPC Consortium. *UPC Language Specifications V1.2*, 2005.

[W3C07] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/soap12-part1/, 2007.

[YSP+98] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, e Alexander Aiken. Titanium: A High-performance Java Dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

Templates

A.1 AtEachPlaceClosure

Listagem A.1: Template AtEachPlaceClosure

```
public static class $AtEachPlace extends Closure {
    private static final long serialVersionUID = 1L;
    private static final short $_serialization_id =
        x10.x10rt.DeserializationDispatcher
        .addDispatcher(
            x10.x10rt.DeserializationDispatcher
            .ClosureKind.CLOSURE_KIND_SIMPLE_ASYNC,
            $AtEachPlace.class);

    public static final x10.rtt.RuntimeType<$AtEachPlace> $RTT =
        x10.rtt.StaticVoidFunType.<$AtEachPlace>make(
    /* base class */$AtEachPlace.class, /* parents */new x10.rtt.Type[] {
        x10.core.fun.VoidFun_0_0.$RTT, x10.rtt.Types.OBJECT });

    public x10.rtt.RuntimeType<?> $getRTT() {
      return $RTT;
    }

    public static x10.x10rt.X10JavaSerializable $_deserialize_body(
        $AtEachPlace $_obj, x10.x10rt.X10JavaDeserializer $deserializer)
        throws java.io.IOException {
```

```
      $_obj.here$place = $deserializer.readInt();
      $_obj.clock$barrier = (x10.lang.Clock) $deserializer.readRef();
#DESERIALIZER
      return $_obj;
    }

    public static x10.x10rt.X10JavaSerializable $_deserializer(
        x10.x10rt.X10JavaDeserializer $deserializer)
        throws java.io.IOException {

      $AtEachPlace $_obj = new $AtEachPlace((java.lang.System[]) null);
      $deserializer.record_reference($_obj);
      return $_deserialize_body($_obj, $deserializer);

    }

    public short $_get_serialization_id() {
      return $_serialization_id;
    }

    public void $_serialize(x10.x10rt.X10JavaSerializer $serializer)
        throws java.io.IOException {
      $serializer.write(this.here$place);
      if (clock$barrier instanceof x10.x10rt.X10JavaSerializable) {
        $serializer.write((x10.x10rt.X10JavaSerializable)
            this.clock$barrier);
      } else {
        $serializer.write(this.clock$barrier);
      }
#SERIALIZER
    }

    // constructor just for allocation
    public $AtEachPlace(final java.lang.System[] $dummy) {
      super($dummy);
    }

    public void $apply() {
      final int nThreads$0 = x10.lang.Runtime.NTHREADS - 1;
      int thread$t = 0;

      #DISTRIBUTE

      for (; true;) {
        final boolean hasMoreThreads$0 = thread$t <= nThreads$0;
```

```
        if (!hasMoreThreads$0) {
          break;
        }

        final boolean has$barrier = #BARRIER;

        #DIVIDEARRAYPERTHREAD

        if(has$barrier) {
          x10.lang.Runtime.runAsync__0$1x10$lang$Clock$2(
            x10.core.ArrayFactory.<x10.lang.Clock> makeArrayFromJavaArray(
                    x10.lang.Clock.$RTT,
                    new x10.lang.Clock[] { this.clock$barrier }),
            ((x10.core.fun.VoidFun_0_0) (new #CLASSNAME.$ExecParallel(
#ARGUMENTS,
                    thread$t,
                    here$place,
                    (java.lang.Class<?>) null))));
        }
        else {
          x10.lang.Runtime
              .runAsync(((x10.core.fun.VoidFun_0_0) (new
                  #CLASSNAME.$ExecParallel(
#ARGUMENTS,
                    thread$t,
                    here$place,
                    (java.lang.Class<?>) null))));
        }
        thread$t++;
      }
    }

#GLOBALVARS
    public int here$place;
    public x10.lang.Clock clock$barrier;

    public $AtEachPlace(#PARAMETERS, int here$place, x10.lang.Clock
        clock$barrier, java.lang.Class<?> $dummy0) {
      {
#INITGLOBALVARS
        this.here$place = here$place;
        this.clock$barrier = clock$barrier;
      }
    }
  }
```

A.2 CreateShared

Listagem A.2: Template CreateShared

```
final x10.array.Array<#TYPE> #ARRAY = new x10.array.Array<#TYPE>(
    (java.lang.System[]) null, x10.rtt.NamedType.<#TYPE>make(
          "#TYPE", /* base class */#TYPE.class
          , /* parents */ new x10.rtt.Type[] {x10.rtt.Types.OBJECT}
          ));

#ARRAY.x10$lang$Object$$init$S();

final x10.array.RectRegion1D region$#ARRAY =new
   x10.array.RectRegion1D((java.lang.System[]) null);

region$#ARRAY.$init(0, #SIZE - 1);

final x10.array.Region myReg$#ARRAY = region$#ARRAY;

#ARRAY.region = myReg$#ARRAY;
#ARRAY.rank = 1;
#ARRAY.rect = true;
#ARRAY.zeroBased = true;
#ARRAY.rail = true;
#ARRAY.size = #SIZE;
#ARRAY.layout_min0 = #ARRAY.layout_stride1 = #ARRAY.layout_min1 = 0;
#ARRAY.layout = null;

final x10.core.IndexedMemoryChunk<#TYPE> chunk$#ARRAY =
   x10.core.IndexedMemoryChunk.<#TYPE>
    allocate(x10.rtt.NamedType.<#TYPE>make(
          "#TYPE", /* base class */#TYPE.class
          , /* parents */ new x10.rtt.Type[] {x10.rtt.Types.OBJECT}
          ), #SIZE, true);

#ARRAY.raw = chunk$#ARRAY;

final x10.core.GlobalRef<x10.array.Array<#TYPE>> #SHAREDNAME = new
   x10.core.GlobalRef<x10.array.Array<#TYPE>>(
    x10.rtt.ParameterizedType.make(x10.array.Array.$RTT,
        x10.rtt.NamedType.<#TYPE>make(
          "#TYPE", /* base class */#TYPE.class
          , /* parents */ new x10.rtt.Type[] {x10.rtt.Types.OBJECT}
          )), #ARRAY,
    (x10.core.GlobalRef.__0x10$lang$GlobalRef$$T) null);
```

A.3 GetSharedClosure

Listagem A.3: Template GetSharedClosure

```
public static class #SHAREDCLOSURENAME extends x10.core.Ref implements
    x10.core.fun.Fun_0_0, x10.x10rt.X10JavaSerializable {
  private static final long serialVersionUID = 1L;
  private static final short $_serialization_id =
      x10.x10rt.DeserializationDispatcher
      .addDispatcher(
          x10.x10rt.DeserializationDispatcher.
          ClosureKind.CLOSURE_KIND_NOT_ASYNC,
          #SHAREDCLOSURENAME.class);

  public static final x10.rtt.RuntimeType<#SHAREDCLOSURENAME> $RTT =
      x10.rtt.StaticFunType.<#SHAREDCLOSURENAME>make(
  /* base class */#SHAREDCLOSURENAME.class, /* parents */new
      x10.rtt.Type[] {
      x10.rtt.ParameterizedType.make(x10.core.fun.Fun_0_0.$RTT,
          x10.rtt.Types.INT), x10.rtt.Types.OBJECT });

  public x10.rtt.RuntimeType<?> $getRTT() {
    return $RTT;
  }

  public static x10.x10rt.X10JavaSerializable $_deserialize_body(
      #SHAREDCLOSURENAME $_obj, x10.x10rt.X10JavaDeserializer
          $deserializer)
      throws java.io.IOException {

    x10.core.GlobalRef g = (x10.core.GlobalRef) $deserializer.readRef();
    $_obj.g = g;
    return $_obj;

  }

  public static x10.x10rt.X10JavaSerializable $_deserializer(
      x10.x10rt.X10JavaDeserializer $deserializer)
      throws java.io.IOException {

    #SHAREDCLOSURENAME $_obj = new
        #SHAREDCLOSURENAME((java.lang.System[]) null);
    $deserializer.record_reference($_obj);
    return $_deserialize_body($_obj, $deserializer);

  }
```

```
  public short $_get_serialization_id() {
    return $_serialization_id;
  }

  public void $_serialize(x10.x10rt.X10JavaSerializer $serializer)
      throws java.io.IOException {

    if (g instanceof x10.x10rt.X10JavaSerializable) {
      $serializer.write((x10.x10rt.X10JavaSerializable) this.g);
    } else {
      $serializer.write(this.g);
    }
  }

  // constructor just for allocation
  public #SHAREDCLOSURENAME(final java.lang.System[] $dummy) {
    super($dummy);
  }

  public #TYPE $apply$G() {
    return $apply$O();
  }

  public #TYPE $apply$O() {

    final x10.array.Array<#TYPE> array = this.g.$apply$G();

    final x10.core.IndexedMemoryChunk<#TYPE> chunk = array.raw;

    final #TYPE res = ((#TYPE[]) chunk.value)[0];

    return res;
  }

  public x10.core.GlobalRef<x10.array.Array<#TYPE>> g;

  public #SHAREDCLOSURENAME(
      final x10.core.GlobalRef<x10.array.Array<#TYPE>> g,
      java.lang.Class<?> $dummy0) {
    {
      this.g = g;
    }
  }

}
```

A.4 DivideArrayPerPlace

Listagem A.4: Template DivideArrayPerPlace

```
final int region$#ARRAY = #ARRAY.length;
final int numPlaces$#ARRAY = x10.lang.Place.numPlaces$O();
final int perPlace$#ARRAY = region$#ARRAY / numPlaces$#ARRAY;
final int restPlaces$#ARRAY = region$#ARRAY % numPlaces$#ARRAY -
    where$place.id > 0 ? 1 : 0;
final int offset$#ARRAY = java.lang.Math.min(where$place.id,
        region$#ARRAY % numPlaces$#ARRAY);
final int begin$#ARRAY = where$place.id * perPlace$#ARRAY +
    offset$#ARRAY;
final int end$#ARRAY = begin$#ARRAY + perPlace$#ARRAY
        + restPlaces$#ARRAY - 1;

final #TYPE #DISTARRAY = java.util.Arrays.copyOfRange(#ARRAY,
    begin$#ARRAY, end$#ARRAY + 1);
```

A.5 DivideArrayPerThread

Listagem A.5: Template DivideArrayPerThread

```
final int subRegion$#ARRAY = #ARRAY.length;
final int numThreads$#ARRAY = x10.lang.Runtime.NTHREADS;
final int perThread$#ARRAY = subRegion$#ARRAY / numThreads$#ARRAY;
final int restThreads$#ARRAY = subRegion$#ARRAY % numThreads$#ARRAY
    - thread$t > 0 ? 1 : 0;
final int subOffset$#ARRAY = java.lang.Math.min(thread$t,
    subRegion$#ARRAY % numThreads$#ARRAY);
final int first$#ARRAY = thread$t * perThread$#ARRAY
      + subOffset$#ARRAY;
final int last$#ARRAY = first$#ARRAY + perThread$#ARRAY
     + restThreads$#ARRAY - 1;

final #TYPE #DISTARRAY = java.util.Arrays.copyOfRange(this.#ARRAY,
        first$#ARRAY, last$#ARRAY + 1);
```

A.6 ExecParallelClosure

Listagem A.6: Template ExecParallelClosure

```
public static class $ExecParallel extends Closure {
    private static final long serialVersionUID = 1L;
    private static final short $_serialization_id =
        x10.x10rt.DeserializationDispatcher
        .addDispatcher(
            x10.x10rt.DeserializationDispatcher.
            ClosureKind.CLOSURE_KIND_SIMPLE_ASYNC,
            $ExecParallel.class);

    public static final x10.rtt.RuntimeType<$ExecParallel> $RTT =
        x10.rtt.StaticVoidFunType.<$ExecParallel>make(
    /* base class */$ExecParallel.class, /* parents */new x10.rtt.Type[] {
        x10.core.fun.VoidFun_0_0.$RTT, x10.rtt.Types.OBJECT });

    public x10.rtt.RuntimeType<?> $getRTT() {
      return $RTT;
    }

    public static x10.x10rt.X10JavaSerializable $_deserialize_body(
        $ExecParallel $_obj, x10.x10rt.X10JavaDeserializer $deserializer)
        throws java.io.IOException {

      if (x10.runtime.impl.java.Runtime.TRACE_SER) {
        java.lang.System.out
            .println("X10JavaSerializable: $_deserialize_body() of "
                + $ExecParallel.class + " calling");
      }
      $_obj.here$place = $deserializer.readInt();
      $_obj.thread$t = $deserializer.readInt();
#DESERIALIZER
      return $_obj;

    }

    public static x10.x10rt.X10JavaSerializable $_deserializer(
        x10.x10rt.X10JavaDeserializer $deserializer)
        throws java.io.IOException {

      $ExecParallel $_obj = new $ExecParallel((java.lang.System[]) null);
      $deserializer.record_reference($_obj);
      return $_deserialize_body($_obj, $deserializer);
```

```
    }

    public short $_get_serialization_id() {

      return $_serialization_id;

    }

    public void $_serialize(x10.x10rt.X10JavaSerializer $serializer)
        throws java.io.IOException {
      $serializer.write(this.here$place);
      $serializer.write(this.thread$t);
#SERIALIZER
    }

    // constructor just for allocation
    public $ExecParallel(final java.lang.System[] $dummy) {
      super($dummy);
    }

    public void $apply() {
      //long start = System.currentTimeMillis();
      //System.out.println("Init "+ here$place + " Thread("+
          thread$t+")");
#PARALLELBODY
      //long end = System.currentTimeMillis() - start;
      //System.out.println(here$place + " Thread("+ thread$t+") time:" +
          (double)end/1000);
    }

#GLOBALVARS
    public int thread$t;
    public int here$place;

    public $ExecParallel(#PARAMETERS, int thread$t, int here$place,
        java.lang.Class<?> $dummy0) {
      {
#INITGLOBALVARS
        this.thread$t = thread$t;
        this.here$place = here$place;
      }
    }

  }
```

A.7 GetSharedClosure

Listagem A.7: Template GetSharedClosure

```
public static class #SHAREDCLOSURENAME extends x10.core.Ref implements
    x10.core.fun.Fun_0_0, x10.x10rt.X10JavaSerializable {
  private static final long serialVersionUID = 1L;
  private static final short $_serialization_id =
      x10.x10rt.DeserializationDispatcher
       .addDispatcher(
           x10.x10rt.DeserializationDispatcher.
           ClosureKind.CLOSURE_KIND_NOT_ASYNC,
           #SHAREDCLOSURENAME.class);

  public static final x10.rtt.RuntimeType<#SHAREDCLOSURENAME> $RTT =
      x10.rtt.StaticFunType.<#SHAREDCLOSURENAME>make(
  /* base class */#SHAREDCLOSURENAME.class, /* parents */new
      x10.rtt.Type[] {
       x10.rtt.ParameterizedType.make(x10.core.fun.Fun_0_0.$RTT,
           x10.rtt.Types.INT), x10.rtt.Types.OBJECT });

  public x10.rtt.RuntimeType<?> $getRTT() {
    return $RTT;
  }

  public static x10.x10rt.X10JavaSerializable $_deserialize_body(
      #SHAREDCLOSURENAME $_obj, x10.x10rt.X10JavaDeserializer
          $deserializer)
      throws java.io.IOException {

    x10.core.GlobalRef g = (x10.core.GlobalRef) $deserializer.readRef();
    $_obj.g = g;
    return $_obj;

  }

  public static x10.x10rt.X10JavaSerializable $_deserializer(
      x10.x10rt.X10JavaDeserializer $deserializer)
      throws java.io.IOException {

    #SHAREDCLOSURENAME $_obj = new
        #SHAREDCLOSURENAME((java.lang.System[]) null);
    $deserializer.record_reference($_obj);
    return $_deserialize_body($_obj, $deserializer);

  }
```

```
  public short $_get_serialization_id() {
    return $_serialization_id;
  }

  public void $_serialize(x10.x10rt.X10JavaSerializer $serializer)
      throws java.io.IOException {

    if (g instanceof x10.x10rt.X10JavaSerializable) {
      $serializer.write((x10.x10rt.X10JavaSerializable) this.g);
    } else {
      $serializer.write(this.g);
    }
  }

  // constructor just for allocation
  public #SHAREDCLOSURENAME(final java.lang.System[] $dummy) {
    super($dummy);
  }

  public #TYPE $apply$G() {
    return $apply$O();
  }

  public #TYPE $apply$O() {

    final x10.array.Array<#TYPE> array = this.g.$apply$G();

    final x10.core.IndexedMemoryChunk<#TYPE> chunk = array.raw;

    final #TYPE res = ((#TYPE[]) chunk.value)[0];

    return res;
  }

  public x10.core.GlobalRef<x10.array.Array<#TYPE>> g;

  public #SHAREDCLOSURENAME(
      final x10.core.GlobalRef<x10.array.Array<#TYPE>> g,
      java.lang.Class<?> $dummy0) {
    {
      this.g = g;
    }
  }

}
```

A.8 ApplyReduction

Listagem A.8: Template ApplyReduction

```
    #TYPE#DIMENSIONS ret$result;
    if(x10.lang.Place.numPlaces$O() > 1) {
      #TYPE#DIMENSIONS[] tmp$array = new #TYPE[x10.lang.Runtime.NTHREADS
          * x10.lang.Place.numPlaces$O()]#DIMENSIONS;
      final x10.array.Array<#TYPE#DIMENSIONS> tmp$results =
          array$resultsdist.$apply$G();
      for(int i = 0; i < tmp$array.length; i++)
        tmp$array[i] = tmp$results.$apply$G(i);
      ret$result = #REDUCTION.reduce(tmp$array);
    } else
      ret$result = #REDUCTION.reduce(array$results);
    return ret$result;
```

A.9 SOMDClass

Listagem A.9: Template SOMDClass

```
class #CLASSNAME extends SOMDEnqueuer<#RETURNTYPE> {
  protected static final long serialVersionUID = 1L;

  private static final short $_serialization_id =
      x10.x10rt.DeserializationDispatcher
       .addDispatcher(
           x10.x10rt.DeserializationDispatcher.
           ClosureKind.CLOSURE_KIND_NOT_ASYNC,
           #CLASSNAME.class);

  public static final x10.rtt.RuntimeType<#CLASSNAME> $RTT =
      x10.rtt.NamedType.<#CLASSNAME>make(
       "#CLASSNAME", /* base class */#CLASSNAME.class, /* parents */
       new x10.rtt.Type[] { x10.rtt.Types.OBJECT });

#RETURNGLOBALVAR
#GLOBALVARS

  public static x10.x10rt.X10JavaSerializable $_deserialize_body(
      #CLASSNAME $_obj, x10.x10rt.X10JavaDeserializer $deserializer)
      throws java.io.IOException {

    return $_obj;
  }
```

```
  public static x10.x10rt.X10JavaSerializable $_deserializer(
      x10.x10rt.X10JavaDeserializer $deserializer)
      throws java.io.IOException {

    #CLASSNAME $_obj = new #CLASSNAME((java.lang.System[]) null);
    $deserializer.record_reference($_obj);
    return $_deserialize_body($_obj, $deserializer);
  }

  public short $_get_serialization_id() {
    return $_serialization_id;
  }

  // constructor just for allocation
  public #CLASSNAME(final java.lang.System[] $dummy) {
    super($dummy);
    lock = new ReentrantLock();
    resultQueue = lock.newCondition();
    hasResult = false;
  }

  // creation method for java code (1-phase java constructor)
  public #CLASSNAME(#PARAMETERS) {
    this((java.lang.System[]) null);
    $init(#ARGUMENTSPARAMS);
  }

  // constructor for non-virtual call
  final public #CLASSNAME #CLASSNAME$$init$S(#PARAMETERS) {
    {
#INITGLOBALVARS
    }
    return this;
  }

  // constructor
  public #CLASSNAME $init(#PARAMETERS) {
    return #CLASSNAME$$init$S(#ARGUMENTSPARAMS);
  }

  final public #CLASSNAME #CLASSNAME$$#CLASSNAME$this() {

    return #CLASSNAME.this;
  }
```

```
  public #RETURNTYPE #METHODNAME(#PARAMETERS) {
      //long start = System.currentTimeMillis();
      //System.out.println("Init transform");
#TRANSFORMARRAYSTODIST
      //long end = System.currentTimeMillis() - start;
      //System.out.println("end transform" + (double)end/1000);

    final x10.lang.Clock clock$barrier = x10.lang.Clock.make("");
    {
      x10.lang.Runtime.ensureNotInAtomic();

      final x10.lang.FinishState x10$__var10 =
          x10.lang.Runtime.startFinish();

      try {
        try {
            {
              {
                if (x10.lang.Place.numPlaces$O() == 1) {

                  final int nThreads$0 = x10.lang.Runtime.NTHREADS - 1;
                  final int here$place = x10.lang.Runtime.home().id;
                  int thread$t = 0;

                  for (; true;) {

                    final boolean hasMoreThreads$0 = thread$t <=
                        nThreads$0;

                    if (!hasMoreThreads$0) {
                      break;
                    }

                    final boolean has$barrier = #BARRIER;

                    #DIVIDEARRAYPERTHREAD

                    if (has$barrier) {

                      x10.lang.Runtime
                        .runAsync__0$1x10$lang$Clock$2(
                            x10.core.ArrayFactory
                                .<x10.lang.Clock> makeArrayFromJavaArray(
                                    x10.lang.Clock.$RTT,
                                    new x10.lang.Clock[] { clock$barrier
                                        }),
```

```
                    ((x10.core.fun.VoidFun_0_0) (new
                        #CLASSNAME.$ExecParallel(
                        #ARGUMENTS,
                        thread$t,
                        here$place,
                        (java.lang.Class<?>) null)));
                } else {
                  x10.lang.Runtime
                    .runAsync(((x10.core.fun.VoidFun_0_0) (new
                        #CLASSNAME.$ExecParallel(
                        #ARGUMENTS,
                        thread$t,
                        here$place,
                        (java.lang.Class<?>) null)));
                }
                thread$t++;
              }
            } else {
              final x10.array.Array<x10.lang.Place> p$places =
                x10.lang.Place.getInitialized$places();
              final x10.lang.Sequence<x10.lang.Place> seq$places =
                p$places.sequence();
              final x10.lang.Iterator<x10.lang.Place> iterator$places
                = seq$places.iterator();

              for (; true;) {

                final boolean hasNextPlace =
                  iterator$places.hasNext$O();

                if (!hasNextPlace) {
                  break;
                }

                final x10.lang.Place where$place =
                  iterator$places.next$G();

                #DIVIDEARRAYPERPLACE

                x10.lang.Runtime.runAt(where$place,
                      ((x10.core.fun.VoidFun_0_0) (new
                        #CLASSNAME.$AtEachPlace(#ARGUMENTS,
                        where$place.id, clock$barrier,
                        (java.lang.Class<?>) null)));
              }
            }
```

```
                clock$barrier.drop();
              }
            }
        } catch (x10.core.Throwable $exc$) {
          throw $exc$;
        } catch (java.lang.Throwable $exc$) {
          throw x10.core.ThrowableUtilities
              .convertJavaThrowable($exc$);
        }
      } catch (x10.core.X10Throwable __lowerer__var__10__) {

        x10.lang.Runtime
            .pushException(((x10.core.X10Throwable)
                (__lowerer__var__10__)));

        throw new x10.lang.RuntimeException();
      } finally {
        {
          x10.lang.Runtime
              .stopFinish(((x10.lang.FinishState) (x10$__var10)));
        }
      }
    }
#ADDRESULTTEMPLATE
  }

#SHAREDCLOSURESCLASSES

  //@Override
  public void run() {
    lock.lock();
    #ASSIGNRESULT#METHODNAME(#ARGUMENTSPARAMS);
    hasResult = true;
    resultQueue.signal();
    lock.unlock();
  }

  //@Override
  public SOMDEnqueuer<#RETURNTYPE> init(Object ... objects) {
    return this.$init(#OBJECTSTOARGS);
  }

  //@Override
  public #RETURNTYPE getResult() {
    lock.lock();
```

```
    while (!hasResult)
      try {
        resultQueue.await();
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    lock.unlock();
    #RETURNRESULT
  }
}
```

A.10 SetResultClosure

Listagem A.10: Template SetResultClosure

```
  public static class $Set$Result$Closure extends Closure {
    private static final long serialVersionUID = 1L;
    private static final short $_serialization_id =
        x10.x10rt.DeserializationDispatcher
        .addDispatcher(
            x10.x10rt.DeserializationDispatcher.
            ClosureKind.CLOSURE_KIND_NOT_ASYNC,
            $Set$Result$Closure.class);

    public static final x10.rtt.RuntimeType<$Set$Result$Closure> $RTT =
        x10.rtt.StaticVoidFunType.<$Set$Result$Closure>make(
    /* base class */$Set$Result$Closure.class, /* parents */new
        x10.rtt.Type[] {
        x10.core.fun.VoidFun_0_0.$RTT, x10.rtt.Types.OBJECT });

    public x10.rtt.RuntimeType<?> $getRTT() {
      return $RTT;
    }

    public static x10.x10rt.X10JavaSerializable $_deserialize_body(
        $Set$Result$Closure $_obj, x10.x10rt.X10JavaDeserializer
            $deserializer)
        throws java.io.IOException {

      $_obj.index = $deserializer.readInt();
      $_obj.value = (#TYPE) $deserializer.readObject();
      x10.core.GlobalRef g = (x10.core.GlobalRef) $deserializer.readRef();
      $_obj.array$results = g;
```

```
    return $_obj;
  }

  public static x10.x10rt.X10JavaSerializable $_deserializer(
      x10.x10rt.X10JavaDeserializer $deserializer)
      throws java.io.IOException {

    $Set$Result$Closure $_obj = new
        $Set$Result$Closure((java.lang.System[]) null);
    $deserializer.record_reference($_obj);
    return $_deserialize_body($_obj, $deserializer);

  }

  public short $_get_serialization_id() {
    return $_serialization_id;
  }

  public void $_serialize(x10.x10rt.X10JavaSerializer $serializer)
      throws java.io.IOException {

    $serializer.write(this.index);
    $serializer.writeObject(this.value);
    if (array$results instanceof x10.x10rt.X10JavaSerializable) {
      $serializer.write((x10.x10rt.X10JavaSerializable)
          this.array$results);
    } else {
      $serializer.write(this.array$results);
    }
  }

  // constructor just for allocation
  public $Set$Result$Closure(final java.lang.System[] $dummy) {
    super($dummy);
  }

  public void $apply() {
    final x10.array.Array<#TYPE> array = this.array$results.$apply$G();

    final x10.core.IndexedMemoryChunk<#TYPE> chunk = array.raw;

    ((#TYPE[])chunk.value)[index] = value;
  }

  public x10.core.GlobalRef<x10.array.Array<#TYPE>> array$results;
  public int index;
```

```
    public #TYPE value;

    public $Set$Result$Closure(
        int index, #TYPE value,
        final x10.core.GlobalRef<x10.array.Array<#TYPE>> array$results,
        java.lang.Class<?> $dummy0) {
      {
        this.index = index;
        this.value = value;
        this.array$results = array$results;
      }
    }

  }
```

A.11 SetSharedClosure

Listagem A.11: Template SetSharedClosure

```
  public static class #SHAREDCLOSURENAME extends Closure {
    private static final long serialVersionUID = 1L;
    private static final short $_serialization_id =
        x10.x10rt.DeserializationDispatcher
        .addDispatcher(
            x10.x10rt.DeserializationDispatcher.
            ClosureKind.CLOSURE_KIND_NOT_ASYNC,
            #SHAREDCLOSURENAME.class);

    public static final x10.rtt.RuntimeType<#SHAREDCLOSURENAME> $RTT =
        x10.rtt.StaticVoidFunType.<#SHAREDCLOSURENAME>make(
    /* base class */#SHAREDCLOSURENAME.class, /* parents */new
        x10.rtt.Type[] {
        x10.core.fun.VoidFun_0_0.$RTT, x10.rtt.Types.OBJECT });

    public x10.rtt.RuntimeType<?> $getRTT() {
      return $RTT;
    }

    public static x10.x10rt.X10JavaSerializable $_deserialize_body(
        #SHAREDCLOSURENAME $_obj, x10.x10rt.X10JavaDeserializer
            $deserializer)
        throws java.io.IOException {

      x10.core.GlobalRef g = (x10.core.GlobalRef) $deserializer.readRef();
      $_obj.g = g;
```

```
#DESERIALIZER
      return $_obj;

    }

    public static x10.x10rt.X10JavaSerializable $_deserializer(
        x10.x10rt.X10JavaDeserializer $deserializer)
        throws java.io.IOException {

      #SHAREDCLOSURENAME $_obj = new
          #SHAREDCLOSURENAME((java.lang.System[]) null);
      $deserializer.record_reference($_obj);
      return $_deserialize_body($_obj, $deserializer);

    }

    public short $_get_serialization_id() {
      return $_serialization_id;
    }

    public void $_serialize(x10.x10rt.X10JavaSerializer $serializer)
        throws java.io.IOException {

      if (g instanceof x10.x10rt.X10JavaSerializable) {
        $serializer.write((x10.x10rt.X10JavaSerializable) this.g);
      } else {
        $serializer.write(this.g);
      }
#SERIALIZER
    }

    // constructor just for allocation
    public #SHAREDCLOSURENAME(final java.lang.System[] $dummy) {
      super($dummy);
    }

    public void $apply() {
      #BEGINATOMIC
        final x10.array.Array<#TYPE> array = this.g.$apply$G();

        final x10.core.IndexedMemoryChunk<#TYPE> chunk = array.raw;

        ((#TYPE[])chunk.value)[0] = #EXPRESSION;

      #ENDATOMIC
    }
```

```
#GLOBALVARS
    public x10.core.GlobalRef<x10.array.Array<#TYPE>> g;

    public #SHAREDCLOSURENAME(
        final x10.core.GlobalRef<x10.array.Array<#TYPE>> g,
#PARAMETERS
        java.lang.Class<?> $dummy0) {
      {
#INITGLOBALVARS
        this.g = g;
      }
    }

  }
```